Introduction ### Who is this book for?

This is an intermediate to advanced level book. Readers should have a firm grasp of JavaScript and a basic understanding of React, Express, Node & MongoDB. The book it more implementation than concept focused.

Testing

Given the amount of time needed to write this book vs the amount of time I have, I decided early on to not focus on testing for fear of turning the project of writing this book into something larger than I would be able to complete. That said, automated tests are essential to making good software.

Code Editor

The client application will be using TypeScript. For the server, using TypeScript types from DefinitelyTyped (http://definitelytyped.org/) as well as types that are built into libraries makes coding significantly easier. While you can use any editor you prefer, Visual Studio Code (https://code.visualstudio.com/) is referenced by some of the examples and in my opinion, is the best free choice for using TypeScript.

Mac, Linux & Windows

I work exclusively on Linux (Ubuntu 18.04). As a result, everything in the book will work on Mac and Linux. There are some difference in configuration for getting things to work on Windows. However, due to time constraints I will not be including that information.

Give Ubuntu (Linux) a Try

For those not familiar with Linux I'll mention that doing software development on Linux is a genuine pleasure and a great learning experience that will help you in the server world. In fact, the Express server build in this book will run on Ubuntu, a variation of Linux. I highly recommend giving Ubuntu a try as you everyday development machine.

Spinning up a Linux virtual machine is an easy thing to do and a good learning experience in itself. In my experience with running a Linux VM on Windows, you need about 8 GB RAM. More is better. If you are short of RAM you can dual boot your system.

Async/Await

The code will use async/await. If you are not familiar with them, under the covers they are promises, but have a better syntax. A good reference for learning to use async/await is XXXXX.

Perferences

Software development is full of preferences and from choice of code editor to libraries used, this book very much reflects mine. My preferences may not be any better than someone elses. They may even be worse. However, they have been working well for me in terms of productivity, flexability and robustness.

Choice: DigitalOcean

I like the price and, even before I started using them for hosting, many of my Google Linux questions landed on DigitalOcean's well writting Linux guides.

Choice: AWS S3

Again, I like the price of hosting on AWS S3. Deploying a React application to S3 is also very easy. I currently have about 6 sites hosted there.

Choice: AWS Route 53

Choice: Not putting the client on the same server (i.e., DigitalOcean droplet) as the server (i.e., todo-server)

bla, bla, bla ... do I need to address this
I have another demo app deployed with both the client and server hosted on a DigitalOcean droplet. This works well and is only costing about $5 per month. However, note it is a demo app and is built with the lowest priced/configuration droplet. A production app would require a higher level of resources.

Things Not Addressed

- Secret Management

Setup

Since you will be building locally and deploying to a remote server, both a client and server, as well as running a database locally, there is quite a bit of setup. Much of this book is about setup and configuration. For some, that isn't the fun part, but it is a necessary prerequisite.

Instead of writing many, many pages on how to do installation, I'll be referring you to external resources. All of the technology used in the book already has excellent installation instructions that I couldn't write any better.

Get a Domain

Without your own domain you won't get the full experience. If you don't already have a domain you want to use you can get a free one from freenom (https://www.freenom.com/en/index.html?lang=en). The one I am using for the books example app, klequis-todo.tk, is from freenom.

Point to instructions on the web

Development Machine Setup

Global vs Local Package Installs

Most npm packages can be installed locally or globally. It is best to install packages locally so that if you need a different versions of a package for different projects, you don't need to keep changing your machines global configuration. The only exception needed for this is Nodemon. Install it locally:

```
sudo npm i -g nodemon
```

nodemon

As mentioned, the documentation for installing Node, MongoDB, MongoDB Compass and Visual Studio Code are excellent. Follow the below links in order to install them.

NodeJS & npm

The best option is the LTS version which as of the moment is v10.15.3. Installation instructions for Linux are Installing Node.js via package manager (https://nodejs.org/en/download/package-manager/). Windows and Mac installers are at Downloads (https://nodejs.org/en/download/).

MongoDB Local Install

The production version of our app will use MongoDB (https://mongodb.com) hosted on MongoDB Atlas (https://www.mongodb.com/cloud/atlas). However, for development it is easier to work with MongoDB locally. Install MongoDB on your local machine using these instructions: Install MongoDB Community Edition (https://docs.mongodb.com/manual/administration/install-community/).

Robo 3T

You'll want a way to view the data in MongoDB. Robo 3T (https://robomongo.org/) is good tool.

Alternatively, MongoDB Compass is also good tool. Compass installation instructions are at Download and Install Compass (https://docs.mongodb.com/compass/master/install/)

I like Compass' UI a bit better than Robo 3T. However, Robo 3T allows you to edit documents as JSON and has a full-fledged built-in query editor.

Visual Studio Code (VS Code)

While you can use the code editor of your choice, some of the examples refer to VS Code functionality and/or configuration. If you are not using VS Code, you may have to do some configuration of your editor. Instructions for installing VS Code are at Setting up Visual Studio Code (https://code.visualstudio.com/Docs/setup/setup-overview).

VS Code Extensions

- Bracket Pair Colorizer

- npm Intellisense
- Prettier - Code Formatter
- Visual Studio IntelliCode

VS Code Settings

Update Imports On File Move

> Last I tried it, doesn't work well with absolute imports and adds unwanted imports
> "javascript.updateImportsOnFileMove.enabled": "never",
> "typescript.updateImportsOnFileMove.enabled": "never",

Window Title

"window.title": "${dirty}${folderName} : ${activeEditorShort}"

perttier

> TODO: Editor configuration should have its own section

> TODO: there are likely more settings. Review Prettier doc: https://prettier.io/docs/en/configuration.html

`perttier.config.js`

```
module.exports = {
  tabWidth: 2,
  semi: false,
  singleQuote: true
}
```

> With machine configuration out of the way you are ready to start building the app!

Express Server Part I

We will start with the minimum needed to have a working and deployable REST API.

Install Packages

```
$ npm init -y
$ npm i @babel/runtime body-parser cors express
$ npm i -D @babel/cli @babel/core @babel/node @babel/plugin-transform-runtime
@babel/preset-env @babel/node
$ npm i -D @babel/types @types/node @types/body-parser @types/express
@types/cors
```

TODO: question: why using babel-node?

> Types are not absolutely necessary, but are helpful when writing code and don't get included in the build so are effectively neutral.

Create Server

The entry point for the server will be server/index.js

```
$ mkdir server
$ touch server/index.js
```

> Configuration will be hard-coded for now.

```
// server/index.js

import express from 'express'
import bodyParser from 'body-parser'
import cors from 'cors'

const app = express()

const port = 3030

app.use(cors())
app.use(bodyParser.json())


app.get('/', (req, res) => {
    res.status(200).send({ data: 'hello', error: '' })
})

app.listen(port, () => {
    console.log(`Events API is listening on port ${port}`)
})
```

Add `babel.config.js`

```
$ touch babel.config.js
```

```
// babel.config.js

module.exports = function (api) {
  api.cache(true);

  const presets = [ "@babel/preset-env" ];
  const plugins = [ "@babel/plugin-transform-runtime" ];

  return {
    presets,
    plugins
  };
}
```

First Run

At this point you can run the server from the command line using npx and `babel-node`.

```
$ npx babel-node server/index.js
```

The terminical should print

```
Events API is listening on port 3030
```

Open a browser and type into the address bar

```
localhost:3030
```

In the response you should see

```
{ data: "hello", error: ""}
```

//
//

Add a "start" script to `package.json` and remove the "test" script for now.

```
...


...
```

Ubuntu Server Part I

TODO: Summary
TODO: Step outline

Generating SSH Keys

Use ssh-keygen to generate a new pair of SSH keys. ssh-keygen will save the keys to the current directory. Either change to ~/.ssh or enter the full path at the prompt.
- t is for type and rsa is the type to be used

```
$ cd ~/.ssh
$ ssh-keygen -t rsa
```

You will be prompted for a file name. Enter a name and press enter.

> WARN: If you are not in the /.ssh folder you need to enter the full path.

```
klequis@klequis-pc:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/klequis/.ssh/id_rsa): ssh-todo-
server-6
```

Leave the passphrase blank and press enter

```
Enter passphrase (empty for no passphrase):
```

Leave blank and press enter

```
Enter same passphrase again:
```

The result will be similar to this:

```
Your identification has been saved in ssh-todo-server-6.
Your public key has been saved in ssh-todo-server-6.pub.
The key fingerprint is:
SHA256:L+fO...4 joe@joes-pc
The key's randomart image is:
+---[RSA 2048]----+
|                .|
|           .o.  o|
|            += = |
|          o...Oo+|
|        S   +=o!%|
|         . o +B#@|
|        . oA+.oBO|
|         - .o .o.|
|         .+  .. .|
+----[SHA256]-----+
```

Your new key is in ~/.ssh. The public key has the `.pub` extension and the private key has not extension. You will upload the public key to your new server.

Back-up your key!

Create Droplet

1. Click Create > Droplets
2. Under Distributions choose Ubuntu > 18.04 x64

3. Choose a plan: Standard
4. $5/mo
5. Choose Region

    - Choose the that is closest to your users. Since this is a tutorial app, you can choose the region closest to you.

6. Select additional options > IPv6
7. Select additional options

- IPv6

1. Add your SSH keys

- Open ssh-todo-server-6.pub
- Copy its contents
- Click 'New SSH Key'
- Paste into the dialog
- Give the same name as it is on your local machine
- Click 'Add SSH Key'

1. Choose a host name: todo-server-6
2. Click Create

Create and Configure Ubuntu Server

Log in as root

Since a SSH key was added to the machine you are able to login as root. From a terminal on your local machine, log in as the root user

```
ssh root@your_server_ip
```

- You will see the message - type 'yes' and hit enter

```
The authenticity of host '206.189.77.123 (206.189.77.123)' can't be established.
ECDSA key fingerprint is SHA256:K2rtC0an9u6UPpEMwEcfUqGyjWT1ckCpCLKKjJAQ56o.
Are you sure you want to continue connecting (yes/no)?
```

The command promp should now look like this:

```
root@todo-server-1:~#
```

Create a New User

You can call your user anytning you want. I'm going to name mine 'doadmon'.

```
adduser doadmin
```

You will be asked to enter a new password for the user.

Next you will be asked a number of questions. Fill-in these values as you see fit.

Grant Administrative Privileges

Then grant this user administrative privileges

```
usermod -aG sudo doadmin
```

Set Up a Firewall

The utility UFW (Uncomplicated Firewall) is already installed on Ubuntu. For more information on UFW see the UFW User Manual (http://manpages.ubuntu.com/manpages/bionic/en/man8/ufw.8.html)

If you check UFW's status you will see it is currently disabled

```
root@todo-server-1:~# ufw status
Status: inactive
```

Some applications such as OpenSSH register themselves with UFW when installed. UFW can only manage applications that are registered with it.

To see which apps are currently registered

```
root@todo-server-1:~# ufw app list
Available applications:
  OpenSSH
```

Enable/allow connection via SSH

```
ufw allow OpenSSH
```

Then enable ufw

```
ufw enable
```

You will see the message

```
Command may disrupt existing ssh connections. Proceed with operation (y|n)?
```

Type y
'y' and hit enter. The response should be

```
Firewall is active and enabled on system startup
```

Now check the status again

```
root@todo-server-1:~# ufw status
Status: active
```

Enable External SSH for Your New User

TODO: what does this command do? Copy ~/.ssh from root to /home/doadmin?

```
rsync --archive --chown=doadmin:doadmin ~/.ssh /home/doadmin
```

Test you new user

Before logging out of the root account, test your new user by opening a new terminal window and logging in as the new user. Since this is the first time you are logging in you will be asked for the users password.

```
ssh doadmin@<ipaddress>
```

From this point forward, you should do all your work on the server with this new user.

Installing Nginx

On same tutorial but step 4

https://www.digitalocean.com/community/tutorials/how-to-set-up-a-node-js-application-for-production-on-ubuntu-18-04

The server is currently only running on local host.

> TODO: The way the server block was setup in the previous step probably isn't necessary. Likely can setup this way first.
> Also have to wonder of ../klequis-todo.tk/html is necessary

After this step the server is available from the command line

```
$ sudo apt update
$ sudo apt install nginx
```

Allow Access Through Firewall

```
$ sudo ufw app list
$ sudo ufw allow 'Nginx HTTP'
$ sudo ufw status
```

> TODO: Question: Although 'Nginx HTTP' was enabled above. I think it should be 'Nginx HTTPS' to allow https traffice only.

Check Status

```
systemctl status nginx
```

Then in the browser

```
http://your-server-ip-address
```

Should see 'Welcome to nginx'

Todo: Link to Nginx commands

Setup a Server Block

```
sudo mkdir -p /var/www/api.klequis-todo.tk/html
sudo chown -R $USER:$USER /var/www/api.klequis-todo.tk/html
```

You may also need to adjust the permissions with

```
$ sudo chmod -R 755 /var/www/api.klequis-todo.tk
```

Create a HTML page to test

```
$ nano /var/www/api.klequis-todo.tk/html/index.html
```

Make the contents

```
<html>
    <head>
        <title>api.klequis-todo.tk</title>
    </head>
    <body>
        <h1>api.klequis-todo.tk is working!</h1>
    </body>
</html>
```

Create the Server Block

```
$ cd /etc/nginx/sites-available
$ sudo nano api.klequis-todo.tk
```

Make the contents

```
server {
        listen 80;
        listen [::]:80;

        root /var/www/api.klequis-todo.tk/html;
        index index.html index.htm index.nginx-debian.html;

        server_name api.klequis-todo.tk www.api.klequis-todo.tk;

        location / {
                try_files $uri $uri/ =404;
        }
}
```

To enable the server block, create a link to it from /sites-enabled

```
sudo ln -s /etc/nginx/sites-available/api.klequis-todo.tk /etc/nginx/sites-
enabled
```

Increase the hash_bucket_size

```
$ sudo nano /etc/nginx/nginx.conf
```

Uncomment the line (or add it if it isn't there)

```
...
http {
  ...
  server_names_hash_bucket_size 64;
  ...
}
...
```

Check configuration for errors

```
$ sudo nginx -t
```

Restart Nginx

```
$ sudo systemctl restart nginx
```

Now visit the site from a browser

> WARN: If visiting the site from the browser doesn't work at all, try restarting the server.

> WARN: It is possible that when visiting your site via ipaddress you will continue to get directed to the default page even though you did everything correctly. Try clearing your browsers cache. If that doesn't work then delete the default site:

```
$ sudo rm /etc/nginx/sites-enabled/default
$ sudo rm /etc/nginx/sites-available/default
$ sudo systemctl restart nginx
```

Deploy Express Server

> Older text for this is in file zz.03.08.create-test-express-server.md

Creating Test Express Server

Next we will deploy the minimal-express-server to test the machine build as well as the additional steps of setting-up PM2 & Nginx as a reverse proxy server.

Building it

Since the server is using some JavaScript features that are not unsupported by node we need to compile the code to an early version of JavaScript which will be es5 ().

Making the build script

- considered a bash script but ...
- need to add package rimraf & fs-extra

deploy it

- log into server as doadmin

```
git clone https://github.com/klequis/minimal-express-server.git app
```

build it

Write a bash script some doc (https://www.linux.com/learn/writing-simple-bash-script)

```

```

Output

```
Events API server is listening on port 3030
```

To test the server, open another terminal on your server and use 'curl' to call it

```
curl http://localhost:3030
```

In the console you should see

```
Response from todo-server
```

Server Build Script

In this section we will using a NodeJS script to create a build script for the server. This script will work for both the current minimal install as well as future increments (?with some modification/additions). (or: as more features are added to the Express server the script will be modified to accommodate them.)

> TODO: all about the script

```
/*
  - use fs-extra it has a convenient 'move' method whcih native Node 'fs' does
not.
  - use rimraf to delete a non-empty directory which can be done with Node but
not as easily.
*/



const cp = require('child_process')
cp.execSync('npm i -D fs-extra rimraf', {stdio:[0,1,2]})


const path = require('path')
const fs = require('fs-extra')
const rimraf = require('rimraf')
const execSync = require('child_process').execSync

// final path for app
const appPath = path.normalize(`${__dirname}/../app`)

// remove ../app it it already exists
if (fs.existsSync(appPath)) {
  rimraf.sync(appPath, {}, function () { console.log("done"); });
}

// npm i for repo
// for `stdio` see:
https://nodejs.org/api/child_process.html#child_process_options_stdio
execSync('npm i', {stdio:[0,1,2]})

// build app
execSync('npm run build', {stdio:[0,1,2]})

// move dist files to ../app
fs.move(`${__dirname}/dist`, appPath)

function l(msg, value) {
  if (value === undefined) {
    console.log(msg)
  } else {
    console.log(msg, value)
  }
}
```

Deploy & Test

Ubuntu Server Part II

Before the server can be accessed externally there are a few more steps to complete:

1. Using PM2
2. Setting Up a Server Block on Nginx
3. Using Nginx as a Reverse Proxy
4. Securing Nginx with Let's Encrypt

Using PM2

PM2 is ...

PM2 will ...

In Brief

Install

```
// Install
$ sudo npm install pm2@latest -g
// Start
$ pm2 start app/server
// Startup
pm2 startup systemd
// Run command from output of last command
$ sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup
systemd -u doadmin --hp /home/doadmin
// Save process list and corresponding environments
$ pm2 save
// Start service with systemctl
$ sudo systemctl start pm2-doadmin
// Check status of systemmd unit
$ systemctl status pm2-doadmin
```

> TODO: The above are the steps. Not clear that they work. Need to try on new server and get info from PM2 documentation.

> TODO: Make decision of showing output. It is really really long.

Installing PM2 globally

If your server is still running go back to the terminal where you started it and press `ctl-c` to stop it.

```
cd
$ sudo npm install pm2@latest -g
```

Run todo-server using `pm2`

```
$ pm2 start app/server
```

Output

```
                        -------------

__/\\\\\\\\\\\\\____/\\\\____________/\\\\____/\\\\\\\\\_____
 _\/\\\/////////\\\_\/\\\\\\________/\\\\\\__/\\\///////\\\___
  _\/\\\_______\/\\\_\/\\\//\\\____/\\\//\\\_\///______\//\\\__
   _\/\\\\\\\\\\\\\/__\/\\\\///\\\/\\\/_\/\\\___________/\\\/___
    _\/\\\/////////____\/\\\__\///\\\/___\/\\\________/\\\//_____
     _\/\\\_____________\/\\\____\///_____\/\\\_____/\\\//________
      _\/\\\_____________\/\\\_____________\/\\\___/\\\/___________
       _\/\\\_____________\/\\\_____________\/\\\__/\\\\\\\\\\\\\\\_
        _\///______________\///______________\///__\///////////////__


                          Runtime Edition

        PM2 is a Production Process Manager for Node.js applications
                     with a built-in Load Balancer.

                Start and Daemonize any application:
                $ pm2 start app.js

                Load Balance 4 instances of api.js:
                $ pm2 start api.js -i 4

                Monitor in production:
                $ pm2 monitor

                Make pm2 auto-boot at server restart:
                $ pm2 startup

                To go further checkout:
                http://pm2.io/


                        -------------

[PM2] Spawning PM2 daemon with pm2_home=/home/doadmin/.pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /home/doadmin/todo-server/server in fork_mode (1 instance)
[PM2] Done.
┌──────────┬────┬─────────┬──────┬───────┬────────┬─────────┬────────┬─────┬───
────────┬─────────┬──────────┐
│ App name │ id │ version │ mode │ pid   │ status │ restart │ uptime │ cpu │
mem       │ user     │ watching │
├──────────┼────┼─────────┼──────┼───────┼────────┼─────────┼────────┼─────┼───
────────┼─────────┼──────────┤
│ server   │ 0  │ 1.0.0   │ fork │ 16082 │ online │ 0       │ 0s     │ 0%  │
25.0 MB   │ doadmin │ disabled │
└──────────┴────┴─────────┴──────┴───────┴────────┴─────────┴────────┴─────┴───
────────┴─────────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app
```

TODO: the below does what

```
pm2 startup systemd
```

output

```
[PM2] Init System found: systemd
[PM2] To setup the Startup Script, copy/paste the following command:
sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd
-u doadmin --hp /home/doadmin
```

Run the command from the output above, which will result in output similiar to this:

```
[PM2] Init System found: systemd
Platform systemd
Template
[Unit]
Description=PM2 process manager
Documentation=https://pm2.keymetrics.io/
After=network.target

[Service]
Type=forking
User=doadmin
LimitNOFILE=infinity
LimitNPROC=infinity
LimitCORE=infinity
Environment=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:/usr/local/games:/snap/bin:/usr/bin:/bin:/usr/local/sbin:/usr/local/b
in:/usr/sbin:/usr/bin
Environment=PM2_HOME=/home/doadmin/.pm2
PIDFile=/home/doadmin/.pm2/pm2.pid
Restart=on-failure

ExecStart=/usr/lib/node_modules/pm2/bin/pm2 resurrect
ExecReload=/usr/lib/node_modules/pm2/bin/pm2 reload all
ExecStop=/usr/lib/node_modules/pm2/bin/pm2 kill

[Install]
WantedBy=multi-user.target

Target path
/etc/systemd/system/pm2-doadmin.service
Command list
[ 'systemctl enable pm2-doadmin' ]
[PM2] Writing init configuration in /etc/systemd/system/pm2-doadmin.service
[PM2] Making script booting at startup...
[PM2] [-] Executing: systemctl enable pm2-doadmin...
Created symlink /etc/systemd/system/multi-user.target.wants/pm2-doadmin.service
→ /etc/systemd/system/pm2-doadmin.service.
[PM2] [v] Command successfully executed.
+---------------------------------------+
[PM2] Freeze a process list on reboot via:
$ pm2 save

[PM2] Remove init script via:
$ pm2 unstartup systemd
```

Now use curl to test the server again

```
$ curl http://localhost:3030
```

Once again you should see

```
Response from todo-server
```

We want PM2 to load and start todo-server whenever the Ubuntu server boots. Use the below command to make that happen

```
pm2 startup systemmd
```

Output

```
[PM2] Init System found: systemd
-----------------------------------------------------------
 PM2 detected systemd but you precised systemmd
 Please verify that your choice is indeed your init system
 If you arent sure, just run : pm2 startup
-----------------------------------------------------------
[PM2] To setup the Startup Script, copy/paste the following command:
sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemmd
-u doadmin --hp /home/doadmin
```

Run the command at the end of the output

```
sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemmd
-u doadmin --hp /home/doadmin
```

TODO: I messed-up here. Think I forgot to run the above command and ran pm2 save and tried to use systemctl (below). PM2 instructued me to use the line

```
sudo env PATH=$PATH:/usr/bin /usr/lib/node_modules/pm2/bin/pm2 startup systemd
-u doadmin --hp /home/doadmin
```

Which is the same as above
Here is the output

```
[PM2] Init System found: systemd
Platform systemd
Template
[Unit]
Description=PM2 process manager
Documentation=https://pm2.keymetrics.io/
After=network.target

[Service]
Type=forking
User=doadmin
LimitNOFILE=infinity
LimitNPROC=infinity
LimitCORE=infinity
Environment=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games:/usr/local/games:/snap/bin:/usr/bin:/bin:/usr/local/sbin:/usr/local/b
in:/usr/sbin:/usr/bin
Environment=PM2_HOME=/home/doadmin/.pm2
PIDFile=/home/doadmin/.pm2/pm2.pid
Restart=on-failure

ExecStart=/usr/lib/node_modules/pm2/bin/pm2 resurrect
ExecReload=/usr/lib/node_modules/pm2/bin/pm2 reload all
ExecStop=/usr/lib/node_modules/pm2/bin/pm2 kill

[Install]
WantedBy=multi-user.target

Target path
/etc/systemd/system/pm2-doadmin.service
Command list
[ 'systemctl enable pm2-doadmin' ]
[PM2] Writing init configuration in /etc/systemd/system/pm2-doadmin.service
[PM2] Making script booting at startup...
[PM2] [-] Executing: systemctl enable pm2-doadmin...
Created symlink /etc/systemd/system/multi-user.target.wants/pm2-doadmin.service
→ /etc/systemd/system/pm2-doadmin.service.
[PM2] [v] Command successfully executed.
+---------------------------------------+
[PM2] Freeze a process list on reboot via:
$ pm2 save

[PM2] Remove init script via:
$ pm2 unstartup systemd
```

Running via systemctl still didn't work. Rebooting solved the problem.

Tell PM2 to save the current list of proceses to manage

```
$ pm2 save
```

Next uses `systemctl` to start PM2
Stop PM2 if it is already running

```
$ pm2 stop server
```

Start with `systemctl`

```
sudo systemctl start pm2-doadmin
```

> You can learn more about PM2 at the PM2 website (https://pm2.io/). Note that PM2 has 3 products. We are using PM2 Runtime (https://pm2.io/doc/en/runtime/overview/) which is also the link to the relevant documentation.
> For more essential PM2 commands see PM2 Runtime Command Sheatsheet (https://pm2.io/doc/en/runtime/features/commands-cheatsheet/)

Setting-up Server Blocks on Nginx

> Per my testing, for a Node server, you do not need to have var/www/klequis-todo.tk/html setup for a Node/Express server.

Create Domain

- api.klequis-todo.tk
- add @ record
- add www record

```
sudo nano /etc/nginx/sites-available/api.klequis-todo.tk
```

> TODO: Contents of file

```
server {
        listen 80;
        listen [::]:80;

        root /var/www/api.klequis-todo.tk/html;
        index index.html index.htm index.nginx-debian.html;

        server_name api.klequis-todo.tk www.api.klequis-todo.tk;

        location / {
                proxy_pass http://localhost:3030;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection 'upgrade';
                proxy_set_header Host $host;
                proxy_cache_bypass $http_upgrade;
        }
}
```

Link it

```
sudo ln -s /etc/nginx/sites-available/api.klequis-todo.tk /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl restart nginx
```

Using Nginx as a Reverse Proxy

Currently todo-server can only be accessed from the server it is running on. To make todo-server accessible to the client application will will setup Nginx as a reverse proxy server.

> TODO: What is a reverse proxy

Edit the server block for api.todo-server.tk

```
$ cd /etc/nginx/sites-available
$ sudo nano api.todo-server.tk
```

Replace the existing `location` block with the following

> CAUTION: Use spaces not tabs. Nginx will consider tabs a syntax error

```
proxy_pass http://localhost:3030;
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection 'upgrade';
proxy_set_header Host $host;
proxy_cache_bypass $http_upgrade;
```

TODO: what does the above do?

The complete file should look like this

```
server {

root /var/www/api.klequis-todo.tk/html;
        index index.html index.htm index.nginx-debian.html;

        server_name api.klequis-todo.tk www.api.klequis-todo.tk;

        location / {
            proxy_pass http://localhost:3030;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }

    listen [::]:443 ssl ipv6only=on; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/api.klequis-todo.tk/fullchain.pem; #
managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/api.klequis-todo.tk/privkey.pem;
# managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

}
server {
    if ($host = api.klequis-todo.tk) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


        listen 80;
        listen [::]:80;

        server_name api.klequis-todo.tk www.api.klequis-todo.tk;
    return 404; # managed by Certbot


}
```

Check the syntax of your edits

```
$ sudo nginx -t
```

Output

```
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

Restart Nginx

```
$ sudo systemctl restart nginx
```

Now go to a browser and enter the address for your server

```
https://api.klequis-todo.tk
```

The page should show

TODO: picture 'response from todo-server'

Wow, that was a lot of work but it feels good to successfully have a server up and running. We can now focus on building out the actual server.

Secure Nginx With Let's Encrypt

Add the Certbot Repository & Install It

```
$ sudo add-apt-repository ppa:certbot/certbot
$ sudo apt install python-certbot-nginx
```

Remove HTTP access and add HTTPS access

```
sudo ufw allow 'Nginx Full'
sudo ufw delete allow 'Nginx HTTP'
```

Check Status

```
$ sudo ufw status
```

The result should be

```
Status: active

To                         Action      From
--                         ------      ----
OpenSSH                    ALLOW       Anywhere
Nginx Full                 ALLOW       Anywhere
OpenSSH (v6)               ALLOW       Anywhere (v6)
Nginx Full (v6)            ALLOW       Anywhere (v6)
```

SNAPSHOT: 670dcc64-71c1-11e9-8056-d7554b508c91

TODO: above says HTTP was removed, but the status 'Nginx Full' seems to mean all/both. Read ufw doc to confirm.

old: TODO: This is currently allowing HTTP & HTTPS traffice in. Once working, try removing HTTP and allowing HTTPS only.

Now get the certificate. This will make entries in the server block so that the certificate is used.

```
$ sudo certbot --nginx -d api.klequis-todo.tk
```

Enter your email at the prompt

```
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Plugins selected: Authenticator nginx, Installer nginx
Enter email address (used for urgent renewal and security notices) (Enter 'c'
to
cancel):
```

Agree to the terms of service - Press A then Enter

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Please read the Terms of Service at
https://letsencrypt.org/documents/LE-SA-v1.2-November-15-2017.pdf. You must
agree in order to register with the ACME server at
https://acme-v02.api.letsencrypt.org/directory
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(A)gree/(C)ancel:Y
```

You will also be asked

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Would you be willing to share your email address with the Electronic Frontier
Foundation, a founding partner of the Let's Encrypt project and the non-profit
organization that develops Certbot? We'd like to send you email about our work
encrypting the web, EFF news, campaigns, and ways to support digital freedom.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(Y)es/(N)o:
```

We only want HTTPS traffic so choose 2 then Enter

```
Obtaining a new certificate
Performing the following challenges:
http-01 challenge for api.klequis-todo.tk
Waiting for verification...
Cleaning up challenges
Deploying Certificate to VirtualHost /etc/nginx/sites-enabled/api.klequis-
todo.tk

Please choose whether or not to redirect HTTP traffic to HTTPS, removing HTTP
access.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1: No redirect - Make no further changes to the webserver configuration.
2: Redirect - Make all requests redirect to secure HTTPS access. Choose this
for
new sites, or if you're confident your site works on HTTPS. You can undo this
change by editing your web server's configuration.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Select the appropriate number [1-2] then [enter] (press 'c' to cancel):
```

```
Redirecting all traffic on port 80 to ssl in /etc/nginx/sites-
enabled/api.klequis-todo.tk


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Congratulations! You have successfully enabled https://api.klequis-todo.tk

You should test your configuration at:
https://www.ssllabs.com/ssltest/analyze.html?d=api.klequis-todo.tk
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

IMPORTANT NOTES:
 - Congratulations! Your certificate and chain have been saved at:
   /etc/letsencrypt/live/api.klequis-todo.tk/fullchain.pem
   Your key file has been saved at:
   /etc/letsencrypt/live/api.klequis-todo.tk/privkey.pem
   Your cert will expire on 2019-07-25. To obtain a new or tweaked
   version of this certificate in the future, simply run certbot again
   with the "certonly" option. To non-interactively renew *all* of
   your certificates, run "certbot renew"
 - Your account credentials have been saved in your Certbot
   configuration directory at /etc/letsencrypt. You should make a
   secure backup of this folder now. This configuration directory will
   also contain certificates and private keys obtained by Certbot so
   making regular backups of this folder is ideal.
 - If you like Certbot, please consider supporting our work by:

   Donating to ISRG / Lets Encrypt:   https://letsencrypt.org/donate
   Donating to EFF:                    https://eff.org/donate-le
```

Go back to the browser where you had api.klequis-todo.tk and refresh. You should see the URL change to https://api.klequis-todo.tk and the browsers security indicator should be positive.

Finally, test the renewal process by doing a dry run

```
sudo certbot renew --dry-run
```

```
Saving debug log to /var/log/letsencrypt/letsencrypt.log

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Processing /etc/letsencrypt/renewal/api.klequis-todo.tk.conf
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Cert not due for renewal, but simulating renewal for dry run
Plugins selected: Authenticator nginx, Installer nginx
Renewing an existing certificate
Performing the following challenges:
http-01 challenge for api.klequis-todo.tk
Waiting for verification...
Cleaning up challenges

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
new certificate deployed with reload of nginx server; fullchain is
/etc/letsencrypt/live/api.klequis-todo.tk/fullchain.pem
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
** DRY RUN: simulating 'certbot renew' close to cert expiry
**          (The test certificates below have not been saved.)

Congratulations, all renewals succeeded. The following certs have been renewed:
  /etc/letsencrypt/live/api.klequis-todo.tk/fullchain.pem (success)
** DRY RUN: simulating 'certbot renew' close to cert expiry
**          (The test certificates above have not been saved.)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

IMPORTANT NOTES:
 - Your account credentials have been saved in your Certbot
   configuration directory at /etc/letsencrypt. You should make a
   secure backup of this folder now. This configuration directory will
   also contain certificates and private keys obtained by Certbot so
   making regular backups of this folder is ideal.
```

/etc/nginx/sites-available/klequis-todo.tk
before
?

after this is old but leave for now

server {
listen 80;
listen [::]:80;

```
    root /var/www/klequis-todo.tk/html;
    index index.html index.htm index.nginx-debian.html;

    server_name klequis-todo.tk www.klequis-todo.tk;

    location / {
            try_files $uri $uri/ =404;
    }

listen [::]:443 ssl ipv6only=on; # managed by Certbot
listen 443 ssl; # managed by Certbot
ssl_certificate /etc/letsencrypt/live/klequis-todo.tk/fullchain.pem; # managed
by Certbot
ssl_certificate_key /etc/letsencrypt/live/klequis-todo.tk/privkey.pem; #
managed by Certbot
include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot
```

}

React Client Part I

Creating a Minimal Client

Building a Minimal Client

Use [Create React App (https://facebook.github.io/create-react-app/)](https://facebook.github.io/create-react-app/) to make the initial application.

```
$ npx create-react-app todo-client
```

Modify sr/App.js to match below

Mods

- Convert component to Class
- Add state
- Add `componentDidMount` with async
- Use [`fetch` (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) to call the API
- Put returned data in state
- Delete from body ... stuff ...
- Add to body (shows object in state)

```
import React from "react";
import logo from "./logo.svg";
import "./App.css";

const codeStyle = {
  color: 'white',
  textAlign: 'left'
}
class App extends React.Component {

  state = {
    data: undefined
  }
  async componentDidMount() {
    const r1 = await fetch("https://api.klequis-todo.tk");
    const data = await r1.json()
    this.setState({ data: data })
  }

  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <div>
            <pre style={codeStyle}>{JSON.stringify(this.state.data, null, 4)}
</pre>
          </div>
        </header>
      </div>
    );
  }
}

export default App;
```

Certificate for Client

TODO: Tried to do this again and the certificate status staid in in Pending status for over 20 minutes. Could it be that I didn't have NameCheap setup first? Don't think so.

TODO: But a certificate is not needed until later because we don't want to use CloudFront until after a lot of the dev work on the client is done.

TODO: However, however, it seems to take a long time for a certificate to validate. You cannot create the certificate before there is a bucket so I'm thinking about earlier in the book to setup a bucket with an html page so that the certificate validation can be complete once the reader gets to the part of the book where it is needed.

Get a certificate for your domain

Prerequisites

- A domain (example will use trivalleycoders.org)
- A bucket you app files
- Bucket is configured with public access
- A DNS setup for your domain. This example uses Route 53
- Any subdomains you want to use. We will use the www subdomain.

Go to the [Certificate Manager (https://console.aws.amazon.com/acm/home?region=us-east-1#/)](https://console.aws.amazon.com/acm/home?region=us-east-1#/)

- You must use the N. Virginia (us-east-1) region which is linked above

- Click 'Request a certificate'

- Select 'Request a public certificate'

- 'Request a certificate'

Step 1 Add domain names

- add trivalleycoders.org
- add www.trivalleycoders.org
  Click 'Next'

Step 2 Select validation method

- Select 'DNS validation'
- Click 'Review'

Step 3 Review

- Definitely make sure it is correct to avoid pain later
- Click 'Confirm and request'

Step 4 Validation
Shortly after the page loads the 'Validation status' column will say 'Pending validation'.

- Click the arrow on the left side of the first domain to open the details.

- Click 'Record record in Route 53'

- A modal dialog appears. Click 'Create'

    - You should see a green 'Success' box but the 'Validation status' may still be 'Pending validation'. Check back a bit later

- Repeat the above X steps for the second domain
- Go to [Route 53 (https://console.aws.amazon.com/route53)](https://console.aws.amazon.com/route53) and navigate to your hosted zone
    - You should see the two CNAMEs added by the certificate validation process
- Go back to the Certificate Manager home page and expand the trivalleycoders.org certificate. Both domains will now have a 'Validation status' of 'Success'. They they are not, wait a few minutes and try again.

Deploy Client to S3

> TODO: OK, not sure of the order yet but I tried getting a new certificate and didn't get the Create record in Route 53 button. Maybe this is because I didn't have a hosted zone setup for the bucket.

Setup a Hosted Zone (http un-encrypted)

- New hosted zone
- Click 'Create Record Set'
- name: blank - record will have the root domain
- Type: A - IPv4 address
- Alias: Yes
- Alias Target: choose the bucket which will be under 'S3 website endpoints'
- Click 'Create'

> TODO: need to setup an AAAA record as well here (or just for CloudFront)
> Create another A record

- Click 'CreateRecord Set'
- Name: www
- Type: A - IPv4 address
- Alias: Yes
- Alias Target: www.domain.tld
- Click 'Create'

> TODO: need to setup an AAAA record as well here (or just for CloudFront)

1. Login
2. Create a new bucket named kelquis-todo.tk
   2.1. Name and Region

- Set Bucket name: klequis-todo.tk
- Set Region (for me): US West (Oregon)
- Click Next
  2.2. Configure options

- take defaults
- Click Next

> Shoot, the changed this screen :(

2.3. Click Next
2.4. Click Next
2.5. Click Create bucket

?. Grant public access?
Click on the new bucket then click Permissions > Public settings and set all 4 options to true

> TODO: Confirm this is really needed. If you don't do so you can't put in the below (public) bucket policy. I suspect that once the site is up and running, one or more of these should be turned back on.
> Per
> https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/GettingStarted
> You only need to uncheck Block new public ACLs and uploading public objects & Remove public access granted through public ACLs - Give this a try.

3. Set bucket policy & ?
   3.1. Click on the new bucket then click Permissions and then Bucket Policy
   3.2.

> Note the ARN just above the edit area
> Paste in the below and change to your ARN

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::klequis-todo.tk/*"
        }
    ]
}
```

3.3 Click Save

4. Click Properties > Static website hosting and set

- Use this bucket to host a website: checked/true
- Index document: index.html
- Error document: index.html
  4.1. Click Save

5. Create another bucket named www.klequis-todo.tk
   Bucket name: www.klequis-todo.tk
   Region: US West (Oregon)
   Next
   Next
   Next
   Create bucket

6. Click Properties > Static website hosting and set

- Redirect requests: checked/true
- Target bucket or domain: klequis-todo.tk

7. Create test file

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <h1>S3 Bucket Test - <span style="color: green"><b>SUCCESS!</b></span></h1>
</body>
</html>
```

7. Go back to the klequis-todo.tk bucket and upload index.html

- can drag and drop file
  Next
  Manage public permissions: Grant public read access to this object(s)
  Next
  Next
  Next
  Upload

Go back to Properties > Static website hosting and click on the endpoint again. The new page should be displayed.

8. Click Properties > Static website hosting and click the endpoint at the top of the card. The site should open.

9. Build the project and upload files from /build

Create a CloudFront Distribution Using HTTPS

TODO: need to point users to AWS doc for account setup. I don't want to reprint that. Make point of being non-root.

TODO: the core of the documentation is here: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/GettingStarted I'm not all that sure how much I want to include this info in the book or should I point readers to the AWS documentation.

TODO: I set this up for klequis.io but did so from the AWS doc, not form the below.

TODO: Although CF worked in past without default object it must now be set

Prerequisites

- Endpoint from your bucket: Properties > Static website hosting
- A certificate that covers your domains (TODO add link to page)
- Go to the 'cloudFront Distributions' page
- Click 'Create Distribution'
- Under 'Web' click 'Get Started'

Fill out the fields as follows (fields not mentioned stay as is)

Origin Settings

- Origin Domain Name: http://trivalleycoders.org.s3-website-us-west-2.amazonaws.com
    - The endpoint from above
    - The Origin ID will auto-fill

Default Cache Behavior Settings

- Viewer Protocol Policy: Redirect HTTP to HTTPS

Distribution Settings

- Price Class: Use Only U.S., Canada and Europe
    - Since I don't expect users in Asia to be looking at this site I use this option. Choose the option that makes sense for you.
- Alternate Domain Names (CNAMEs): (on sperarate lines trivalleycoders.org, www.trivalleycoders.org
- SSL Certificate: Custom SSL Certificate
    - Click in the edit box and choose your certificate from above

TODO: include instructions for logging?

You will be redirected to the CloudFront Distributions page. The status of your distribution will be 'in Progress' for 15 minutes or so. In the mean time, 'Update Your DNS to use CloudFront'

Updating DNS Records to Point to CloudFront

Click 'Create Distribution'

- Record the 'Domain Name': dyf96x1rifcde.cloudfront.net
- Double check that IPv6 is enabled
- Go to [Route 53 (https://console.aws.amazon.com/route53/home)](https://console.aws.amazon.com/route53/home)
- Click on 'Hosted zones' and then your zone (trivalleycoders.org)
- Select the A record for trivalleycoders.org and change the Alias Target to: dyf96x1rifcde.cloudfront.net
    - You may see your domain in the list under 'CloudFront distributions'. If you do you can select it rather than pasting in the domain.
- Click 'Save'
- Click 'Create Record Set'
- Leave the name blank (this will create a record for the apex domain 'trivalleycoders.org')
- Type: AAAA - IPv6 address
- Alias: Yes
- Alias Target: damifv64aaqx5.cloudfront.net
- Click Save
- Repeat the above 8 steps for the www.trivalleycoders.org A record
- Test you site. All the below patterns should by resolving to https://trivalleycoders.org or https://www.trivalleycoders.org.

Note: You may need to clear your browser's cache

- https://trivalleycoders.org
- apex domain: trivalleycoders.org
- www sub-domain: www.trivalleycoders.org
- http://trivalleycoders.org
- https://trivalleycoders.org
- http://www.trivalleycoders.org

- https://www.trivalleycoders.org

END OF STEPS

[Amazon CloudFront Developer Guide]https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.htm

This article explains how to setup the DNS records: [Routing Traffic to an Amazon CloudFront Web Distribution by Using Your Domain Name (https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-to-cloudfront-distribution.html)](https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-to-cloudfront-distribution.html)

> *Important:* When using an S3 bucket configured as a Website Endpoint for the origin, use the bucket's static hosting endpoint for the origin. Do not use the bucket name.

> Route 53 charges for CNAME queries. It doesn't charge for queries to Alias queries

Maybe not if you are using HTTPS, dee p.95: For using custom domain name see Amazon CloudFront - Developer Guide (pdf) p.59 Using Custom URLs for Files by Adding Alternate Domain Names (CNAMEs)

> *Mistake:* It appears that when you are using Route 53 for DNS you should use an A name. If you are using another DNS provider you should use a CNAME. I was using Route 53 with CNAME so not correct. Source: CloudFront Developer Guide (pdf) page 61.
> *Solution:* Yes, that was the problem. Create the A and AAAA records with the CloudFront domain name

Checkout dig: http://www.kloth.net/services/dig.php as a way of confirming resource record set

Here is the section for using HTTPS: Using HTTPS with CloudFront p. 85

HTTPS

There are 2 parts to https with CloudFront

1. requests between viewers and CloudFront
2. requests between CloudFront and the origin

We are using a 'S3 bucket configured as a website endpoint' as the origin. According to the manual on p.90

> If your Amazon S3 bucket is configured as a website endpoint, you can't configure CloudFront to use HTTPS to communicate with your origin because Amazon S3 doesn't support HTTPS connections in that configuration.

DNS Setup on AWS Route53

What is going on here

Now that you have a server with an IP address you can setup the DNS for it.
If you don't have a domain and would like to get a free one to use with this example, go to [freenom (https://www.freenom.com/en/index.html)](https://www.freenom.com/en/index.html) to get one. I'll be using the domain 'klequis-todo.tk' which I got from freenom.

We will be using Amazon Route 53 as the DNS in the example. There are other ways to setup DNS but since the client will be running on the server and I use Route 53 for a number of other sites, I'll be doing it there in the example.

Summary

- Get ip info
- Create Hosted Zone
- Add sub domain
  ** Leave the rest for after there is a client to deploy

There is more to do for setting up the domain once we have a client deployed. However, for now, this setup will allow you to test the server once it is ready.

No go to the site where you registered your domain. For me that is freenom. Find your domain and its name server settings
Fill in the name servers

-
-

** **Keep this & incorporate later**

Create Hosted Zone

Creating a new Hosted Zone automatically creates NS and SOA records. You need to create record sets for both client and server as listed below.

App	Record Type	domain	Value
Client	A	klequis-todo.tk	CloudFront endpoint
	AAAA	klequis-todo.tk	CloudFront endpoint
	A	www.klequis-todo.tk	CloudFront endpoint
	AAAA	www.klequis-todo.tk	CloudFront endpoint
Server	A	api.klequis-todo.tk	todo-server IPv4 address
	AAAA	api.klequis-todo.tk	todo-server IPv6 address

TODO: QUESTION: Setting-up www.api.klequis-todo.tk seems unnecessary since it will only be called from the client which will never use www.

TODO: Chicken & egg: Should the hosted zone be created with the non-CloudFront addresses and then later modified, or should the setup be done after CloudFront is setup. I guess that depens on if a CF distribution can be cretaed before the Hosted Zone is setup.

Create a record set for the apex domain

When creating a record set for the apex domain, in our case klequis-todo.tk, you leave the 'Name' edit box empty.

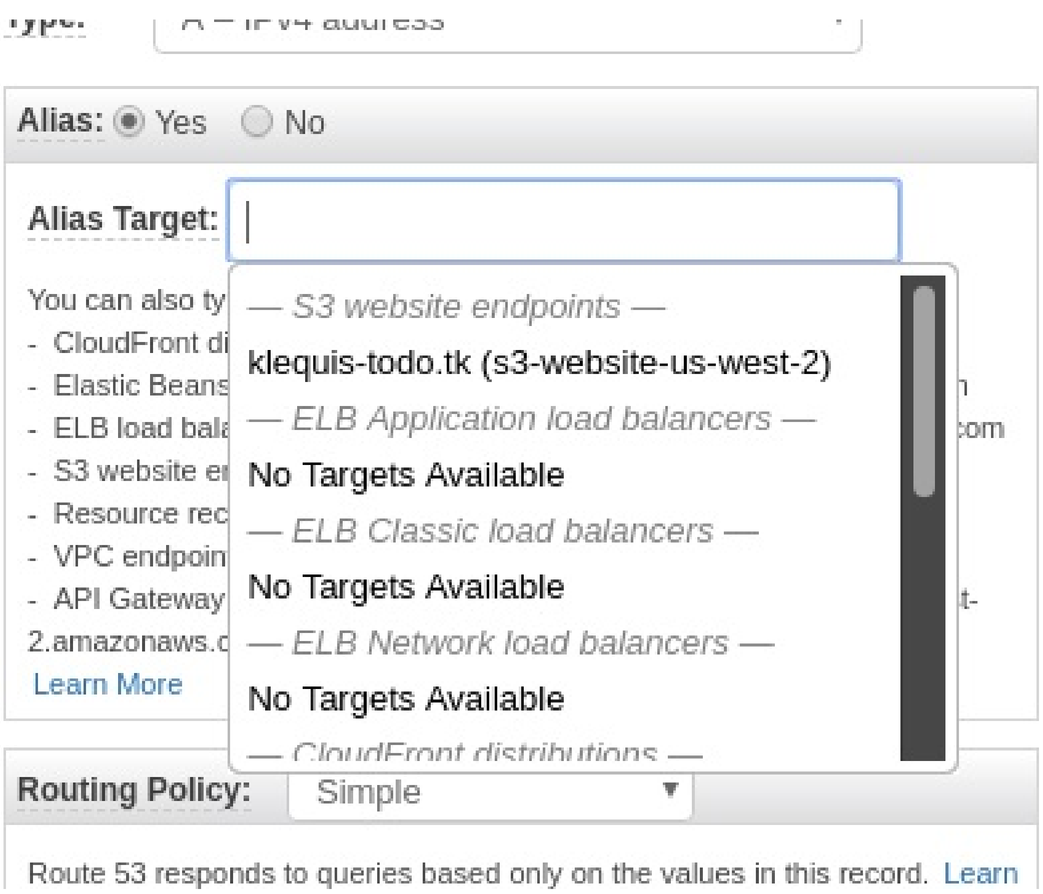
1. Click Create Record Set
    - Name: blank
    - Type: A - IPv4Address
2. Click 'Alias'
    - Alias Target: klequis-todo.tk (s3-website-us-west-2)
3. Other settings are left as the default
    - Routing policy: Simple (default)
    - Evaluate Target Health: No (default)
4. Click Create
5. Repeat steps 1 - 4 but choose 'AAAA - IPv6Address' for 'Type'

Create a record set for a sub domain

When creating a record set for a subdomain you put the subdomain part of the address in the 'Name' edit box. The rest of the address is added for you.

1. Click Create Record Set
    - Name: www
    - Type: A - IPv4Address
2. Click 'Alias'
    - Alias Target: ?

6. Repeat steps 10.1 - 10.4 above for www.klequis-todo.tk



Alias: ● Yes ○ No

Alias Target:

You can also ty
- CloudFront di
- Elastic Beans
- ELB load bala
- S3 website er
- Resource rec
- VPC endpoin
- API Gateway
2.amazonaws.c

Learn More

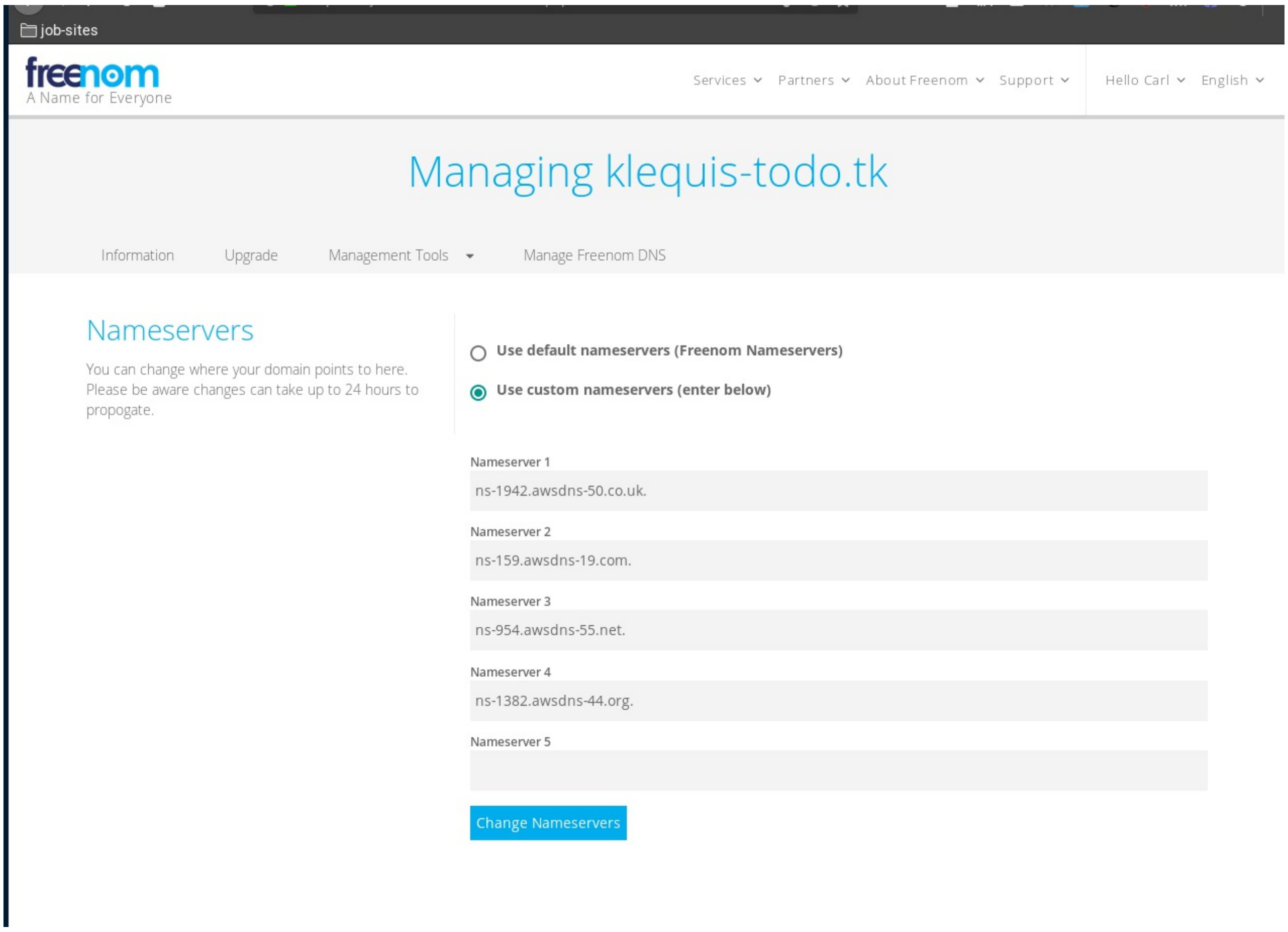
— S3 website endpoints —
klequis-todo.tk (s3-website-us-west-2)
— ELB Application load balancers —
No Targets Available
— ELB Classic load balancers —
No Targets Available
— ELB Network load balancers —
No Targets Available
— CloudFront distributions —

Routing Policy: Simple

Route 53 responds to queries based only on the values in this record. Learn

7. Set Name Servers

klequis-todo.tk is registered on freenom.com. Each registrar has a way for entering name servers. Here is how it looks on freenom



Managing klequis-todo.tk

Nameservers

You can change where your domain points to here. Please be aware changes can take up to 24 hours to propogate.

Use default nameservers (Freenom Nameservers) [✓ Use custom nameservers (enter below)]

Nameserver 1	ns-1942.awsdns-50.co.uk.
Nameserver 2	ns-159.awsdns-19.com.
Nameserver 3	ns-954.awsdns-55.net.
Nameserver 4	ns-1382.awsdns-44.org.
Nameserver 5	

> It can take some time, officially up to 48 hours, for the name server records to get distributed across the internet.

Since AWS Route 53 is managing the DNS for the apex domain (klequis-todo.tk) it also has to manage the sub-domain api.klequis-todo.tk

- Go to Route 53 (https://console.aws.amazon.com/route53)
- Click 'Hosted zones'
- klequis-todo.tk
- Create A record ... 138.197.192.163
- Create AAA record ... 138.197.192.163

MondgoDB

> TODO: Some information about Monogo and references to learn the basics
> TODO: What we will be doing
>
> 1. Setting-up ad MongoDB database and collection
> 2. Wrapping calls to Mongo

About MongoDB queries

If you are unfamiliar with MongoDB queries, a quick overview can be found at CRUD Operations (http://mongodb.github.io/node-mongodb-native/3.2/tutorials/crud/).

In this section we will be using `find` with a query & projection. Some good references for learning about these are:

- Query Documents (https://docs.mongodb.com/manual/tutorial/query-documents/index.html)
- CRUD Operations (http://mongodb.github.io/node-mongodb-native/3.2/tutorials/crud/)
- Projections (http://mongodb.github.io/node-mongodb-native/3.2/tutorials/projections/)

Chaining

MongoDB functions are 'chainable'. For example (pseudo code):

```
const a = find().project().toArray()
```

In this case

- the resutls of `find` are passed to `project`
- the results of `project` are passed to `toArray`
- the results of `toArray` go into the variable a.

> TODO: move this section to 'Formatting Return Values'

Shaping Returned Data

The client application using the API needs to know the shape of the returned data so it can process it properly. This requires the data sent to the client has a consistent shape. We will use the following shape:

```
{
  data: [],
  error: string
}
```

- 'data' is always an array of documents even if it is only one document
- 'error' is the string returned by `e.message`

Creating a MongoDB Database

Since the first database function we will create is 'find', we need something to find. Let's create a database, a collection and one document.

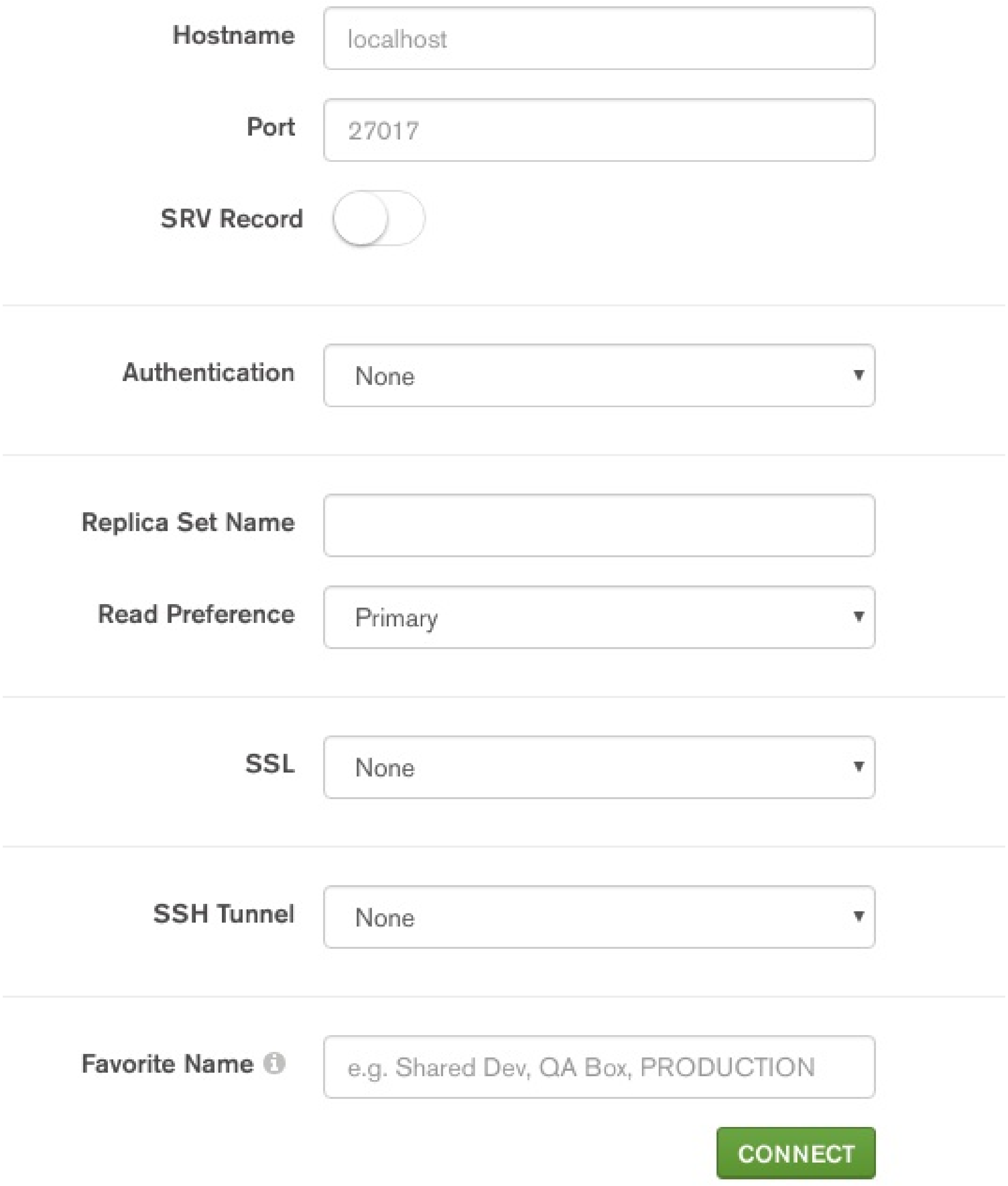
- Database name: todo-dev
- Collection name: todos
- Document structure

```
_id: ObjectID
title: string
complete: boolean
```

- Make sure MongoDB is running on your local machine. If it isn't you can start it (Ubuntu command):

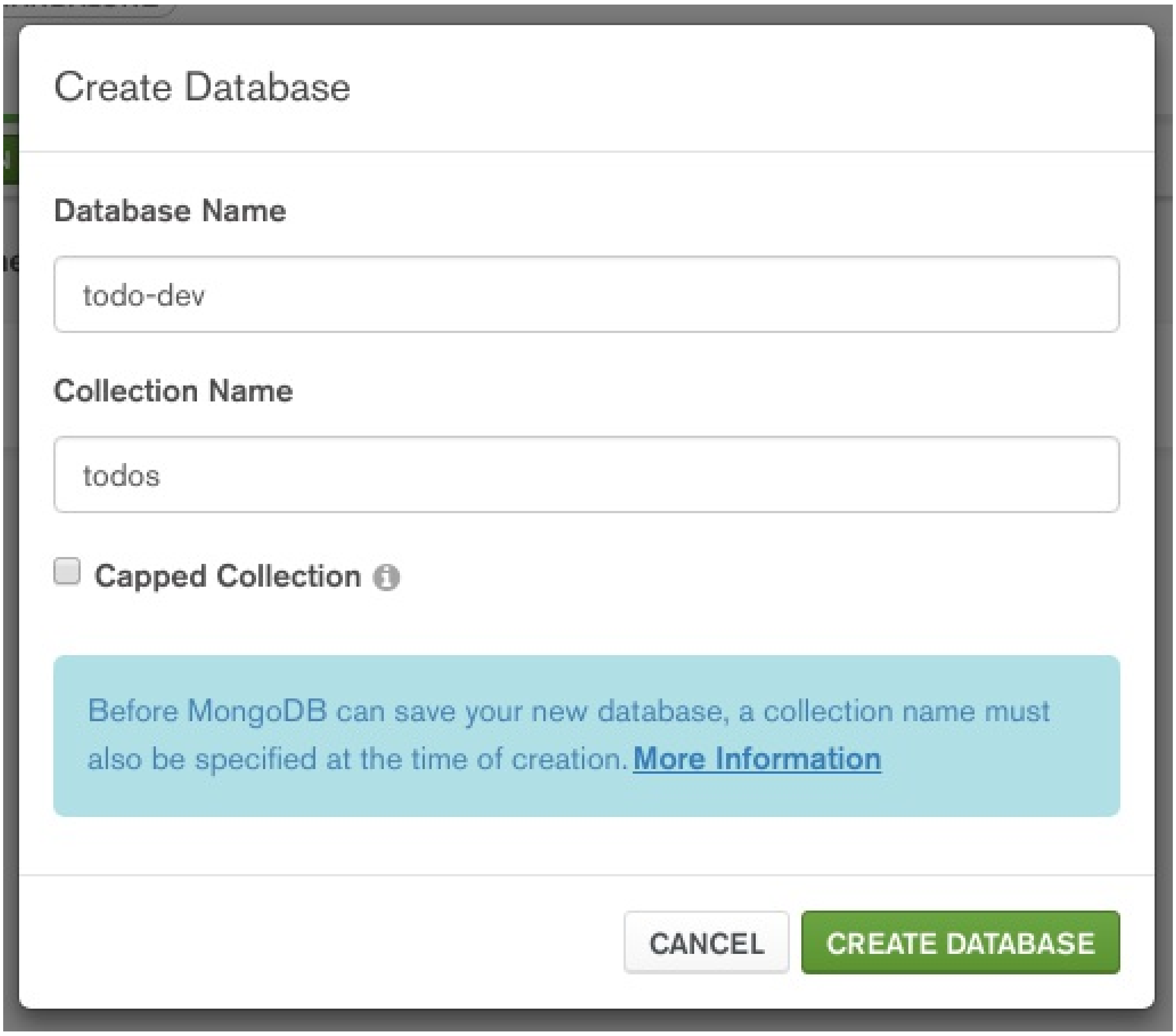
```
sudo service mongod start
```

- Open MongoDB Compass
- Click on 'NEW CONNECTION'
- Fill in the fields as show below



Hostname	localhost
Port	27017
SRV Record	
Authentication	None
Replica Set Name	
Read Preference	Primary
SSL	None
SSH Tunnel	None
Favorite Name	e.g. Shared Dev, QA Box, PRODUCTION

- Click 'CONNECT'
- At the very bottom of the left-hand pannel is a small '+'. Click it to create a new database and collection.


Create Database

Database Name

todo-dev

Collection Name

todos

Capped Collection

Before MongoDB can save your new database, a collection name must also be specified at the time of creation. More Information

Click 'CREATE DATABASE'# Creating the Project

TODO: show the project structure

Project Structure

```
$ mkdir wrapping-calls-to-mongodb
$ cd wrapping-calls-to-mongodb
$ mkdir db
$ cd db
$ touch dbFunctions.js helpers.js index.js
cd ..
mkdir server
touch server/index.js
touch .babelrc .gitignore
mkdir config
touch config/index.js
```

npm Packages

TODO: list of packages and why

TODO: should changes be made to the initial package.json

```
$ npm init -y
$ npm i mongodb @babel/runtime chalk ramda
$ npm i -D @babel/cli @babel/core @babel/node @babel/plugin-transform-runtime
@babel/preset-env chai eslint eslint-plugin-import mocha nodemon supertest
```

Babel Configuration

.babelrc

```
{
  "presets": ["@babel/preset-env"],
  "plugins": [
    "@babel/plugin-transform-runtime",
  ]
}
```

.gitignore

```
dist
node_modules
coverage
logs
log
*.log
app.log.*
npm-debug.log*
pids
*.pid
*.seed
lib-cov
.lock-wscript
.npm
.node_repl_history
.nyc_output
```

Configuration Data

config/index.js

TODO: note- this is not secure

```
const mongoUrl = (env) => {
  if (env === 'test') {
    return 'mongodb://localhost:27017'
  }
  return 'mongodb+srv://todo-db-admin:D92dARWONO0t16uF@todo-cluster0-
ilc7v.mongodb.net/test?retryWrites=true'
}

const dbName = (env) => {
  if (env === 'test') {
    return 'todo-test'
  } else if (env === 'dev') {
    return 'todo-dev'
  }
  return 'todo-prod'
}

const apiRoot = (env)  => {
  if (env === 'prod') {
    return ''
  }
  return 'https://api.klequis-todo.tk'
}

export default {
  mongoUrl: mongoUrl(process.env.NODE_ENV),
  dbName: dbName(process.env.NODE_ENV),
  apiRoot: apiRoot(process.env.NODE_ENV),
  port: 3030
};
```

Helper Functions

db/helpers.js

```
import { ObjectID } from  'mongodb'
import { omit } from 'ramda'

export const objectIdFromHexString = async (hexId) => {
  try {
    return await ObjectID.createFromHexString(hexId)
  }
  catch (e) {
    console.error('ERROR /db/helpers.js.objectidFromHexString', e)
  }
}

export const getObjectId = async (id) => {
  if (ObjectID.isValid(id)) {
    const objId = await objectIdFromHexString(id)

    return objId
  } else {
    throw new Error('ERROR /db/helpers.js.getObjectId', e)
  }
}

export const removeIdProp = (obj) => {
  return omit(['_id'], obj)
}
```

Connecting and Disconnecting

A reference to the client will be stored in the `client` variable and reused each time a call needs to be made. The `connectDB` function will make the connection to the client if it isn't already made, and then return a reference to the database.

Add Imports and `connectDB` to `db/dbFunctions.js`

```
import mongodb, { ObjectID } from 'mongodb'
import { removeIdProp } from './helpers'
import config from '../config'

const MongoClient = mongodb.MongoClient

let client

const connectDB = async () => {
  if (!client) {
    client = await MongoClient.connect(config.mongoUrl, { useNewUrlParser: true
})
  }
  return { db: client.db(config.dbName) }
}
```

Add `close` to `db/dbFunctions.js`

```
export const close = async ()  => {
  if (client) {
    client.close()
  }
  client = undefined
}
```

Add `` to exports in db/index.js

db/index.js

```
export {
  close,
} from './dbFunctions'
```

Formatting Return Values

formatReturn will be used after each call to MongoDB to standardize the way that messages are sent back to the client.

~~~
const formatReturnSuccess = (data)  => {
  return { data: data, error: '' }
}

const formatReturnError = (error)  => {
  return { data: [], error: error.message }
}
```# Logging Errors

> TODO: do you really want to log errors to the console?

```js
const logError = (functionName, error)  => {
  console.error(`Error: dbFunctions.${functionName}`, error.message)
}
~~~

Writing Tests

TODO: I'm unable to find a reference for the return values of MongoDB methods.

In this section you'll get ready to write tests for the MongoDB wrapper functions in db/dbFunctions.js. I highly recommend reading through the Mocha (https://mochajs.org/), Chai (https://www.chaijs.com/) & SuperTest (https://www.npmjs.com/package/supertest) documentation.

Testing Libraries

We will be using Mocha (https://mochajs.org/), Chai (https://www.chaijs.com/) & SuperTest (https://www.npmjs.com/package/supertest)

Mocha

Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

Chai

Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any javascript testing framework.

SuperTest

Supertest is a library made specifically for testing nodejs http servers. It has a very simple syntax that makes the task of testing HTTP calls fairly simple.

Mocha's Hooks

Mocha implements four hooks (https://mochajs.org/#hooks): `before`, `after` `beforeEach` and `afterEach`, which are used to setup preconditions and clean-up after tests.

Hook Scope

The scope of the hook depends upon where it is placed.

Hooks outside of all `describe` blocks

Hooks outside of all `describe` block are called 'root-level hooks'.

- `before`: will run once before all tests
- `after`: will run once after all tests have completed
- `beforeEach`: will run once before each test
- `afterEach`: will run once after each test

Hooks inside a `describe` block

- `before`: will run once before the tests in the `describe` block
- `after`: will run once after the tests in the `describe` block
- `beforeEach`: will run once before each test in the `describe` block
- `afterEach`: will run once after each test in the `describe` block

Let's look at an example of using `before` and `after`

```
before('root before', function rootBefore() {
  console.log('root before')
})

after( async () => {
  console.log('root after')
})

describe('wrapper', () => {
  describe('nested.1', function() {
    before('before nested.1', function() {
      console.log('      before inside nested.1')
    })
    it('test1.1', function() {
    })
    it('test1.2', function() {
    })
    after('after nested.1', function() {
      console.log('      after inside nested.1')
    })
  })
  describe('nested.2', function() {
    before('before nested.2', function() {
      console.log('      before inside nested.2')
    })
    it('test2.1', function() {
    })
    it('test2.2', function() {
    })
    after('after nested.2', function() {
      console.log('      after inside nested.2')
    })
  })
})
```

Running these test produces the output:

```
root before
  wrapper
    nested.1
      before inside nested.1
      ✓ test1.1
      ✓ test1.2
      after inside nested.1
    nested.2
      before inside nested.2
      ✓ test2.1
      ✓ test2.2
      after inside nested.2

root after
```

Note:

- The root `before` runs once before any test test run
- The root `after` run once after all tests have completed
- For nested.1 and nested.2 describe blocks, `before` runs before the tests in the block start and `after` runs following the completion of all tests in the block

Now let's look at the behavior of `beforeEach` and `afterEach`.

```
beforeEach(function rootbeforeEach() {
  console.log('root beforeEach')
})

afterEach( async function() {
  console.log('root afterEach')
})

describe('wrapper', function() {
  describe('nested.1', function() {
    beforeEach(function() {
      console.log('      beforeEach inside nested.1')
    })
    it('test1.1', function() {
    })
    it('test1.2', function() {
    })
    afterEach('after nested.1', function() {
      console.log('      afterEach inside nested.1')
    })
  })
  describe('nested.2', function() {
    beforeEach(function() {
      console.log('      beforeEach inside nested.2')
    })
    it('test2.1', function() {
    })
    it('test2.2', function() {
    })
    after(function() {
      console.log('      afterEach inside nested.2')
    })
  })
})
```

Note

- Every `beforeEach`, at the root level and those inside a describe block, runs before each test
- Every `afterEach`, at the root and those inside a describe block, runs after the completion of each test

Chai

Chai has three assertion styles (https://www.chaijs.com/guide/styles/), Should, Expect and Assert. The choice of interface affects the syntax of the tests. We will be using the Expect style (https://www.chaijs.com/guide/styles/#expect). Both Expect and Should styles are BDD style tests. A reference for the syntax and usage can be found here (https://www.chaijs.com/api/bdd/).

Test Environment Configuration

From project root

```
npm i -D @babel/register
mkdir test
touch test/mocha.opts
mkdir test/dbFunctions
touch test/dbFunctions/dbFunctions.test.js
touch test/dbFunctions/fixture.js
```

Add `@babel/register`

```
npm i -D @babel/register
```

mocha.opts

```
--require @babel/register
--watch
--recursive
```

Add Test Scripts to `package.json`

```
"scripts": {
  "test": "NODE_ENV=test NODE_PATH=./ mocha --require @babel/register",
  "test-watch": "export WATCH='watch' && nodemon --exec 'npm test'"
},
```

Test Data

Our test data is kept in a file name fixture.js
test/dbFunctions/fixture.js

```
export const fourTodos = [
  {
    title: 'first todo',
    completed: false,
  },
  {
    title: 'second todo',
    completed: false,
  },
  {
    title: 'third todo',
    completed: false,
  },
  {
    title: 'fourth todo',
    completed: false,
  },
]

export const oneTodo =
{
  title: 'one single todo',
  completed: false,
}
```

Drop Collection

The first database function `dropCollection` is only used by other tests. The client will never drop a collection. There is no way to test it without functions we haven't written yet, so let's assume it works. There will be some obvious errors in the other tests if it doesn't.

Before adding `dropCollection` we need to add some imports and constants, as well as an `after` hook to the top of the module.

db/dbFunctions.js

```
import { expect } from 'chai'
import { fourTodos } from './fixture'
import {
  dropCollection,
} from 'db'

const collectionName = 'todos'

after(async () => {
  await close()
})
```

The `dropCollection` function takes one parameter, a collection name.

Add `` to `db/dbFunctions.js`

db/dbFunctions.js

~~~~
/**
 *
 * @param {string} collection the name of a collection
 */
export const dropCollection = async (collection) => {
  try {
    const { db } = await connectDB()
    const ret = await db.collection(collection).drop()
    return formatReturnSuccess(ret)
  }
  catch (e) {
    if (e.message = 'ns not found') {
      return true
    } else {
      logError('dropCollection', e)
      return formatReturnError(e)
    }
  }
}
```# Inserting One Document

Add `insertOne` to `db/dbFunctions.js`

```js
/**
 *
 * @param {string} collection the name of a collection
 * @param {object} data a documnet, without _id, to be inserted
 */
export const insertOne = async (collection, data) => {
  try {
    const { db } = await connectDB()
    const ret = await db.collection(collection).insertOne(data)
    return formatReturnSuccess(ret.ops[0])
  }
  catch (e) {
    console.error('ERROR: dbFunctions.insertOne', e)
    return formatReturnError(e)
  }
}
~~~~

Add `insertOne` to exports in `db/index.js`

db/index.js

```
export {
  dropCollection,
  insertOne // added
} from './dbFunctions'
```

Add insertOne to imports in dbFunctions.test.js

```
import {
  dropCollection,
  insertOne // added
} from 'db'
```

Return Value

insertOne returns a large object of which only a small part is needed for our purposes. Here is a shortened version of the returned object.

```
{
  result: { n: 1, ok: 1 },
  connection:
   Connection {
     _events: { ... }
     id: 0,
     options: { ... }
     logger: Logger { className: 'Connection' },
     bson: BSON {},
     tag: undefined,
     maxBsonMessageSize: 67108864,
     port: 27017,
     host: 'localhost',
     socketTimeout: 360000,
     keepAlive: true,
     keepAliveInitialDelay: 300000,
     connectionTimeout: 30000,
     responseOptions: { ... }
     flushing: false,
     queue: [],
     writeStream: null,
     destroyed: false,
     hashedName: '29bafad3b32b11dc7ce934204952515ea5984b3c',
     workItems: [],
     socket: { ... }
     buffer: null,
     sizeOfMessage: 0,
     bytesRead: 0,
     stubBuffer: null,
     ismaster: { ... }
  message: { ... }
  ops: [ { title: 'todo added', _id: 5ce2a7d9d1d0cc3ae9f545fc } ],
  insertedCount: 1,
  insertedId: 5ce2a7d9d1d0cc3ae9f545fc
}
```

The part that will be returned is:

```
ops: [ { title: 'todo added', _id: 5ce2a7d9d1d0cc3ae9f545fc } ]
```

Testing `insertOne`

Since this is the first test to be written, we need to do a little setup of the `dbFunctions.test.js` module

test/dbFunctions.test.js

```
import { expect } from 'chai'
import { fourTodos } from './fixture'
import {
  dropCollection,
  insertOne // added
} from 'db'
```

Now we can write the test

~~~
describe('test insertOne', function() {
  before(async function() {
    await dropCollection(collectionName)
    await insertMany(collectionName, fourTodos)
  })
  // insertOne will only be used for new todos.
  // for new todos, competed is always false and set by the server
  const newData = { title: 'todo added' }
  it('insertOne: should insert new document', async function() {
    const i = await insertOne(collectionName, newData)
    expect(i.data._id).to.be.not.null
    expect(i.data.title).to.equal('todo added')
  })
})
```# Inserting Many Documents

Add `insertMany` to `db/dbFunctions.js`

```js
/**
 *
 * @param {string} collection the name of a collection
 * @param {Array} data  an array of documents, without _id, to be inserted
 */
export const insertMany = async (collection, data) => {
  try {
    const { db } = await connectDB()
    const ret = await db.collection(collection).insertMany(data)
    return formatReturnSuccess(ret.ops)
  }
  catch (e) {
    console.warn('ERROR: dbFunctions.insertMany', e.message)
    return formatReturnError(e)
  }
}
~~~

Add insertMany to exports in db/index.js

db/index.js

```
export {
  dropCollection,
  insertMany // added
  insertOne
} from './dbFunctions'
```

Add insertMany to imports in dbFunctions.test.js

```
import {
  dropCollection,
  insertOne // added
} from 'db'
```

Return Value

The return value of `insertMany` is much smaller than the return value of `insertOne`:

```
{ result: { ok: 1, n: 4 },
  ops:
   [ { title: 'first todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc3 },
     { title: 'second todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc4 },
     { title: 'third todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc5 },
     { title: 'fourth todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc6 } ],
  insertedCount: 4,
  insertedIds:
   { '0': 5ce2abbbb647ec4022322bc3,
     '1': 5ce2abbbb647ec4022322bc4,
     '2': 5ce2abbbb647ec4022322bc5,
     '3': 5ce2abbbb647ec4022322bc6 } }
```

Again, the part that will be returned is:

```
ops:
   [ { title: 'first todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc3 },
     { title: 'second todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc4 },
     { title: 'third todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc5 },
     { title: 'fourth todo',
       completed: false,
       _id: 5ce2abbbb647ec4022322bc6 } ],
```

Testing `insertMany`

db/dbFunctions.test.js

~~~
describe('test insertMany', function() {
  before(async function() {
    await dropCollection(collectionName)
  })
  it('insertMany: should insert 4 todos', async function() {
    const i = await insertMany(collectionName, fourTodos)
    expect(i.data.length).to.equal(4)
  })
})
```# Finding Documents

Add `find` to `db/dbFunctions.js`

```js
/**
 *
 * @param {string} collection the name of a collection
 * @param {object} filter filter criteria
 * @param {object} project a valid projection
 */
export const find = async (collection, filter = {}, project = {}) => {
  try {
    const { db } = await connectDB()
    const ret = await
db.collection(collection).find(filter).project(project).toArray()
    return formatReturnSuccess(ret)
  }
  catch (e) {
    logError('find', e)
    return formatReturnError(e)
  }
}
~~~

Add find to exports in db/index.js

db/index.js

```
export {
  dropCollection,
  find, // added
  insertOne
} from './dbFunctions'
```

Add find to imports in dbFunctions.test.js

```
import {
  dropCollection,
  find,  // added
  insertOne
} from 'db'
```

Return Value

```
[ { _id: 5ce2c64ca2e12f5c5ea0a5fd,
    title: 'first todo',
    completed: false },
  { _id: 5ce2c64ca2e12f5c5ea0a5fe,
    title: 'second todo',
    completed: false },
  { _id: 5ce2c64ca2e12f5c5ea0a5ff,
    title: 'third todo',
    completed: false },
  { _id: 5ce2c64ca2e12f5c5ea0a600,
    title: 'fourth todo',
    completed: false } ]
```

Since the return value of find is simply an array of the inserted documents, the entire return value will be returned.

Testing `find`

test/dbFunctions.test.js

~~~
describe('test find', function() {
  before(async function() {
    console.log('**before**');
    await dropCollection(collectionName)
    await insertMany(collectionName, fourTodos)
  })
  it('find: should return 4 todos', async function() {
    const f = await find(collectionName)
    expect(f.data.length).to.equal(4)
  })
})
```# Find by ID

<!-- To test `findById` we will setup the database by prepopulating it with
todos from our test data and then compare one of the ids of the prepopulated
todos to the one retrieved via `findById` The test is added to
`dbFunctions.test.js`. -->


> TODO: This page was written prior to standardizing the format for dbFunctions
& dbFunctions.test.js. It needs a through review. Some of its content may be
incorrect and/or not needed.

Add `findById` to `db/dbFunctions.js`

```js
/**
 *
 * @param {string} collection the name of a collection
 * @param {string} id a valid _id as string
 * @param {object} project a valid projection
 */
export const findById = async (collection, id, project = {}) => {
  try {
    const { db } = await connectDB()
    const ret = await db.collection(collection).find({ _id: ObjectID(id)
}).project(project).toArray()
    return formatReturnSuccess(ret)
  }
  catch (e) {
    logError('findById', e)
    return formatReturnError(e)
  }
}
~~~

Add findById to exports in db/index.js

```
export {
  ...
  find,
  findById, // added
  insertMany,
  ...
} from './dbFunctions'
```

Add findById to imports in test/dbFunctions.test.js.

```
import {
  ...
  find,
  findById, // added
  insertMany,
  ...
} from 'db'
```

TODO: Review the below

Since the functions needed to setup the database have already been tested, setup will be done inside a before hook. Inside of the hook we need to:

- drop the 'todos' collection to start with a fresh database
- populate test data with inserMany
- save the id of the second todo returned in idToFind
- call findById passing to it idToFind
- confirm the id of the todo returned from findById matches the one in idToFind

Since MongoDB ids come back as objects and two _id objects will never equal each other. The object ids must be converted to strings in order to be compared.<sup>(1)</sup>

TODO: end review

Return Value

```
[ { _id: 5ce2cd6e9851d7613e16b178,
    title: 'first todo',
    completed: false } ]
```

Now we are ready for the findById test. The steps are:

- retrieve the todo with the _id saved above in findById
- check that one todo was returned
- get the _id of the returned todo and convert it to a string
- check that the _id of the returned todo matches the _id in idToFind

Test findById

```
// dbFunctions.test.js

describe('test findById', function() {
  let idToFind = undefined
  before(async function() {
    await dropCollection(collectionName)
    const inserted = await insertMany(collectionName, fourTodos)
    idToFind = inserted.data[0]._id.toString()
  })
  it('findById: should return 1 todo with id of second todo', async function()
{
    const f = await findById('todos', idToFind)
    expect(f.data.length).to.equal(1)
    const idFound = f.data[0]._id.toString()
    expect(idFound).to.equal(idToFind)
  })
})
```

Why are two objects with the same value(s) not equal to each other?

JavaScript, and programming languages in general have 'primitive types' and 'reference types'. Primitive types are values in memory. Object types are 'references' (aka pointers or addresses) of an object in memory.

When comparing two objects, JavaScript compares the references. Therefore:

```
const a = { name: 'joe' }
const b = a
console.log(a === b) // true - they are the same object and therefore the same
references
const c = { name: 'joe' }
console.log(a === c) // false - a & c are two different objects.
```

Find One and Delete

Add `findOneAndDelete`to `db/dbFunctions.js`

db/dbFunctions.js

```
/**
 *
 * @param {string} collection the name of a collection
 * @param {string} id a valid _id as string
 */
export const findOneAndDelete = async (collection, id) => {
  try {
    const { db } = await connectDB()
    const ret = await db.collection(collection).findOneAndDelete({ _id:
ObjectID(id) })

    return formatReturnSuccess(ret.value)
  }
  catch (e) {
    logError('findOneAndDelete', e)
    return formatReturnError(e)
  }
}
```

Add `findOneAndDelete`to exports in `db/index.js`

db/index.js

```
export {
  ...
  findById,
  findOneAndDelete,
  insertMany,
  ...
} from './dbFunctions'
```

Add `findOneAndDelete`to imports in `dbFunctions.test.js`

db/dbFunctions.test.js

```
import {
  ...
  findById,
  findOneAndDelete,
  insertMany,
  ...
} from 'db'
```

Return Value

```
{ lastErrorObject: { n: 1 },
  value:
   { _id: 5ce2d07f1b34ed62e556ed2f,
     title: 'second todo',
     completed: false },
  ok: 1 }
```

Testing `findById`

db/dbFunctions.test.js

```
describe('test findOneAndDelete', function() {
  let idToDelete = undefined
  before(async function() {
    await dropCollection(collectionName)
    const inserted = await insertMany(collectionName, fourTodos)
    idToDelete = inserted.data[1]._id.toString()
  })
  it('findOneAndDelete: should delete 1 of 4 todos', async function() {
    const deleted = await findOneAndDelete(collectionName, idToDelete)
    const idDeleted = deleted.data._id.toString()
    expect(idDeleted).to.equal(idToDelete)
  })
})
```

Find one And Update

Add `` to `db/dbFunctions.js`

db/dbFunctions.js

```
/**
 *
 * @param {string} collection the name of a collection
 * @param {string} id a valid _id as string
 * @param {object} update document properties to be updated such as { title:
'new title', completed: true }
 * @param {boolean} returnOriginal if true, returns the original document
instead of the updated one
 */
export const findOneAndUpdate = async (collection, id, update, returnOriginal =
false) => {
  try {

    // if the filter has the _id prop, remove it
    const cleanedUpdate = removeIdProp(update)
    const { db } = await connectDB()
    const ret = await db.collection(collection).findOneAndUpdate(
      { _id: ObjectID(id) },
      { $set: cleanedUpdate },
      { returnOriginal: returnOriginal }
    )
    return formatReturnSuccess(ret.value)
  }
  catch (e) {
    console.warn('ERROR: dbFunctions.findOneAndUpdate', e)
    return formatReturnError(e)
  }
}
```

Add `` to exports in db/index.js

db/index.js

```
export {
  ...
  findOneAndDelete,
  findOneAndUpdate, // added
  insertMany,
  ...
} from './dbFunctions'
```

Add `` to imports in dbFunctions.test.js

db/dbFunctions.test.js

```
import {
  ...
  findOneAndDelete,
  findOneAndUpdate,
  insertMany,
  ...
} from 'db'
```

Return Value

```
{ lastErrorObject: { n: 1, updatedExisting: true },
  value:
   { _id: 5ce2f3b35a4b466e49cb0d3c,
     title: 'changed title',
     completed: true },
  ok: 1 }
```

Testing `findById`

db/dbFunctions.test.js

```
describe('test findOneAndUpdate', function() {
  const newData = { title: 'changed title', completed: true }
  let idToUpdate = undefined
  before(async function() {
    await dropCollection(collectionName)
    const inserted = await insertMany(collectionName, fourTodos)
    idToUpdate = inserted.data[1]._id.toString()
  })
  it('findOneAndUpdate: should return updated document', async function() {
    const updated = await findOneAndUpdate(collectionName, idToUpdate, newData)
    expect(updated.data._id.toString()).to.equal(idToUpdate)
    expect(updated.data.title).to.equal(newData.title)
    expect(updated.data.completed).to.equal(newData.completed)
  })
```

ZZ Testing Find

There is a bit of a chicken and egg problem with writing the first test. The database must be cleaned-up and then prepopulated before functions that act on the data can be used. To clean-up and prepopulation, functions that are not tested yet must be used. As a result, rather than putting setup in a `before` hook, the setup will be done inside of tests.

The functions used in this group of test are `close` `dropCollection`, `find` & `insertMany`. `dropCollection`. `close`, which is part of clean-up, will not have a test. You will know if close fails if the server fails to shutdown.<sup>(1)</sup>

Imports

We import expect from chai, `fourTodos` from the fixture and the three functions mentioned above from `db/dbFunctions.js`. The test code goes into `test/dbFunctions.test.js`.

```
// dbFunctions.test.js
import { expect } from "chai"
import { fourTodos } from "./fixture"
import {
  close,
  dropCollection,
  find,
  insertMany
} from "../../db"
```

A root-level `after` is used to call `close` on the database once all the tests are completed. This allows the server to shutdown after test completion.

```
// dbFunctions.test.js
...

after( async () => {
  await close()
})
```

Start with a `describe` block with the label 'dbFunctions'

```
// dbFunctions.test.js
...

describe('dbFunctions', () => {

})
```

Before running any tests we want a clean/empty database to start with so that when the test run we know exactly what is in it. to do this use `dropCollection`. `dropCollection` will return ...

```
 { data: true, error: '' }
```

... if the call is successful

> NOTE: It is a bit untypical to clean-up before rather than after the test run. However, I sometimes find it useful to see what is in the database after the tests have completed, so am doing clean-up before the tests run.

```
// dbfunctions.test.js
...

describe('test dropCollection', function() {
  it('dropCollection: should return true', async function() {
    const drop = await dropCollection('todos')
    expect(drop.data).to.be.true
  })
})

...
```

Use `console.log()` often. For example, you can check the return value of `dropCollection` like this:

```
console.log('d', d)
```

This the database needs to be prepopulated using `insertMany`. Since the test data inserted has 4 todos, the test will check that an 4 elements were inserted. Using console.log('d', d) following the call to `insertMany` returns:

```
{
   data: {
     result: { ok: 1, n: 4 },
     ops: [ [Object], [Object], [Object], [Object] ],
     insertedCount: 4,
     insertedIds: {
       '0': 5cc9f3cb5498d8389941cec4,
       '1': 5cc9f3cb5498d8389941cec5,
       '2': 5cc9f3cb5498d8389941cec6,
       '3': 5cc9f3cb5498d8389941cec7
     }
   },
   error: ''
}
```

Given this result, we can test for `data.result.n === 4`

```
// dbFunctions.test.js
...

describe('test insertMany', function() {
  it('insertMany: should insert 4 todos', async function() {
    const i = await insertMany('todos', fourTodos)
    expect(i.data.result.n).to.equal(4)
  })
})

...
```

`find` should also return an array of four elements. The return value of `find` is:

```
{
    data: [
        {
          _id: 5cc9f676323b3539a3603ed3,

          title: 'first todo',
          completed: false
        },
        {

            _id: 5cc9f676323b3539a3603ed4,

            title: 'second todo',

            completed: false
        },
        {
            _id: 5cc9f676323b3539a3603ed5,
            title: 'third todo',
            completed: false
        },
        {
            _id: 5cc9f676323b3539a3603ed6,
            title: 'fourth todo',
            completed: false
        }
    ],
    error: ''
}
```

Given this result we can test that `data.length === 4`

```
// dbFunctions.test.js
...

describe('test find', function() {
  it('find: should return 4 todos', async function() {
    const f = await find('todos')
    console.log('f', f);

    expect(f.data.length).to.equal(4)
  })
})

...
```

Here is the complete module.

```
// dbFunctions.test.js

import { expect } from 'chai'
import { fourTodos } from './fixture'
import {
  close,
  dropCollection,
  find,
  findById,
  insertMany
} from '../../db'

after( async () => {
  await close()
})

describe('dbFunctions', () => {
  // describe('test find', function() {
  describe('test dropCollection', function() {
    it('dropCollection: should return true', async function() {
      const drop = await dropCollection('todos')
      expect(drop.data).to.be.true
    })
  })
  describe('test insertMany', function() {
    it('insertMany: should insert 4 todos', async function() {
      const i = await insertMany('todos', fourTodos)
      expect(i.data.result.n).to.equal(4)
    })
  })
  describe('test find', function() {
    it('find: should return 4 todos', async function() {
      const f = await find('todos')
      console.log('f', f);

      expect(f.data.length).to.equal(4)
    })
  })
})
```

Notes

1. To see the effect of close not working comment out the call to close and run the tests.

```
after( async () => {
  // await close()
})
```

When the have completed you will not be returned to the command prompt. Press <kbd>ctl-c</kbd> to do so.

ZZ The Remaining Tests

Now that we have gone through several tests in detail, I'll proceed through the remainder of the tests more quickly.

Make sure the 3 additional functions are imported

```
import {
  close,
  dropCollection,
  find,
  findById,
  findOneAndDelete, // added
  insertOne, // added
  insertMany,
  findOneAndUpdate // added
} from '../../db'
...
```

findOneAndDelete

The test will check that the _id of the returned document matches the _id we used when calling `findOneAndDelete`. If `findOneAndDelete` has failed, a document will not be returned

```
// dbFunctions.test.js

describe('test findOneAndDelete', function() {
  let idToDelete = undefined
  before(async function() {
    await dropCollection('todos')
    const i = await insertMany('todos', fourTodos)
    idToDelete = i.data[1]._id.toString()

  })
  it('findOneAndDelete: should delete 1 of 4 todos', async function() {
    const d = await findOneAndDelete('todos', idToDelete)
    const idDeleted = d.data._id.toString()
    expect(idDeleted).to.equal(idToDelete)

  })
})
...
```

findOneAndUpdate

Since we have written `findOneAndUpdate` to have `returnOriginal = false` the returned value will be the updated document. Check that the _id is the same and that values in the updated document match the new data sent.

```
// dbFunctions.test.js

describe('test findOneAndUpdate', function() {
  const newData = { title: 'changed title', completed: true }
  let idToUpdate = undefined
  before(async function() {
    await dropCollection('todos')
    const i = await insertMany('todos', fourTodos)
    idToUpdate = i.data[1]._id.toString()
  })
  it('findOneAndUpdate: should return updated document', async function() {
    const u = await findOneAndUpdate('todos', idToUpdate, newData)
    expect(u.data._id.toString()).to.equal(idToUpdate)
    expect(u.data.title).to.equal(newData.title)
    expect(u.data.completed).to.equal(newData.completed)
  })
})
...
```

insertOne

Check that the document returned has a non-null _id and that its title is the one we sent.

```
// dbFunctions.test.js

describe('test insertOne', function() {
  // insertOne will only be used for new todos.
  // for new todos, competed is always false and set by the server
  const newData = { title: 'todo added' }
  it('insertOne: should insert new document', async function() {
    const i = await insertOne('todos', newData )
    expect(i.data._id).to.be.not.null
    expect(i.data.title).to.equal('todo added')
  })
})
```

That's it! All your tests should be passing now. Here is the completed module

```
// dbFunctions.test.js

import { expect } from 'chai'
import { fourTodos } from './fixture'
import {
  close,
  dropCollection,
  find,
  findById,
  findOneAndDelete,
  insertOne,
  insertMany,
  findOneAndUpdate
} from '../../db'
```

```
after( async () => {
  await close()
})

describe('dbFunctions', () => {
  // describe('test find', function() {
  describe('test dropCollection', function() {
    it('dropCollection: should return true', async function() {
      const drop = await dropCollection('todos')
      expect(drop.data).to.be.true
    })
  })
  describe('test insertMany', function() {
    it('insertMany: should insert 4 todos', async function() {
      const i = await insertMany('todos', fourTodos)
      expect(i.data.length).to.equal(4)
    })
  })
  describe('test find', function() {
    it('find: should return 4 todos', async function() {
      const f = await find('todos')
      expect(f.data.length).to.equal(4)
    })
  })

  describe('test findById', function() {
    let idToFind = undefined
    before(async function() {
      await dropCollection('todos')
      const i = await insertMany('todos', fourTodos)
      idToFind = i.data[0]._id.toString()
    })
    it('findById: should return 1 todo with id of second todo', async
function() {
      const f = await findById('todos', idToFind)
      expect(f.data.length).to.equal(1)
      const idFound = f.data[0]._id.toString()
      expect(idFound).to.equal(idToFind)
    })
  })

  describe('test findOneAndDelete', function() {
    let idToDelete = undefined
    before(async function() {
      await dropCollection('todos')
      const i = await insertMany('todos', fourTodos)
      idToDelete = i.data[1]._id.toString()

    })
    it('findOneAndDelete: should delete 1 of 4 todos', async function() {
      const d = await findOneAndDelete('todos', idToDelete)
```

~~~
      const idDeleted = d.data._id.toString()
      expect(idDeleted).to.equal(idToDelete)

    })
  })

  describe('test findOneAndUpdate', function() {
    const newData = { title: 'changed title', completed: true }
    let idToUpdate = undefined
    before(async function() {
      await dropCollection('todos')
      const i = await insertMany('todos', fourTodos)
      idToUpdate = i.data[1]._id.toString()
    })
    it('findOneAndUpdate: should return updated document', async function() {
      const u = await findOneAndUpdate('todos', idToUpdate, newData)
      expect(u.data._id.toString()).to.equal(idToUpdate)
      expect(u.data.title).to.equal(newData.title)
      expect(u.data.completed).to.equal(newData.completed)
    })
  })

  describe('test insertOne', function() {
    // insertOne will only be used for new todos.
    // for new todos, competed is always false and set by the server
    const newData = { title: 'todo added' }
    it('insertOne: should insert new document', async function() {
      const i = await insertOne('todos', newData )
      expect(i.data._id).to.be.not.null
      expect(i.data.title).to.equal('todo added')
    })
  })

})

```# ZZ Wrapping Calls to MongoDB

??? What is this ???
Is it still needed?

The first thing we are going to do is create wrappers to some of the MongoDB
native driver's functions. Writing these functions, along with a small helper
library and standardizing the way that data and errors are returned to the
client will, make our code much more understandable and easier to maintain.

- About MongoDB queries
- Shaping data
- New project
- Install packages
- Create /db
- Create dbFunctions.js, helpers.js & index.js
~~~

~~~
- Import mongodb
- Store reference to MongoClient
- Connection details
- Connection management
- Returning errors
- Returning data
- Implementing `find()`
- Write tests



> TODO: what libs install, what they do and why. Maybe just links to save room
and error?









## db/dbFunctions.js



All our calls to MongoDB will be done through functions in dbFunctions.js.
Start by importing mongodb and getting a reference to the MongoClient.




```js
~~~

server/index.js

```
import { find } from './db'

async function findTest() {
  const todos = await find('todos')
  console.log('todos', todos)
}

findTest()
```

Add start script in package.json

```
"start": "babel-node server/index.js"
```

Give it a try

```
$ npm start
```

In the next section we will write some test for the `fined` function.

Express Server Part II

Outline

We will be creating an Express server implemented as a REST API. The API will read and write data to and from Mongod and server the results to the client.

The major components of the tech stack for the server are

Production

Package	Description
Express (https://www.npmjs.com/package/express)	A web server build on top of NodeJS
MongoDB (https://mongodb.com)	A free and open-source cross-platform document-oriented database program. Classified as a NoSQL database program
[MongoDB Atlas]	A cloud version of MongoDB
MongoDB Node.JS Driver (http://mongodb.github.io/node-mongodb-native/)	The official MongoDB driver
Babel v7 latest (https://babeljs.io/docs/en/next/index.html)	We will be using Babel to transform our ES2015+ code to a backward compatible version of JavaScript.
body-parser (https://www.npmjs.com/package/body-parser)	Node middleware for parsing the body of incoming requests
cors (https://www.npmjs.com/package/cors)	A package for providing a Connect/Express middleware that can be used to enable CORS with various options.

Morgan (https://www.npmjs.com/package/morgan)	HTTP request logger middleware for node.js
Ramda (https://ramdajs.com/)	"A practical functional library for JavaScript programmers." Ramda is similar to Immutiable (https://immutable-js.github.io/immutable-js/docs/#/) but provides more functional approach.

Development

Package	Description
Nodeman (https://www.npmjs.com/package/nodemon)	Simple monitor script for use during development of a node.js app.
Chai (https://www.npmjs.com/package/chai)	Chai is a BDD / TDD assertion library for node and the browser that can be delightfully paired with any javascript testing framework.
Mocha (https://www.npmjs.com/search?q=mocha)	Simple, flexible, fun test framework
Supertest (https://www.npmjs.com/package/supertest)	SuperAgent driven library for testing HTTP servers.

Why not Mongoose?

I'm not recommending avoiding Mongoose. Its merits are obvioułs from its wide-spread use. The primary reason I'm not using it is that when I was using Mongoose for a project I found reasons to go around it. That produced two ways to call the database and thereby more confusing code. I am also more comfortable working at a lower level.

Why Ramda?

When I was first learning React I used a book that introduced Ramda to me. I'm both comfortable and happy with it, and see no motivation for using something else.

Configuration

TODO: seems I have not been using absolute references/imports. Need to implement this from very start.

TODO: seems it would be a good idea to put the config code in 'wrapping-calls-to-mongodb'

TODO: currently in test the server is shutdown via a 'kill' and in prod there is no shutdown. Look here for solution: https://expressjs.com/en/advanced/healthcheck-graceful-shutdown.html

We are going to do this and that

Add packages

```
$ npm i mongodb morgan ramda
$ npm i -D nodemon chai mocha supertest @babel/register
```

TODO:Question: Is chai-http needed?

Configuration

TODO: If I do put config into wrapping-calls-to-mongodb, this part will also be a copy paste

```
$ mkdir config
$ touch config/index.js
```

Don't put config/index.js in git

```
// .gitignore
config/index.js
```

```
// config/index.js

const mongoUrl = () => {
  const env = process.env.NODE_ENV
  if (env === 'test') {
    return 'mongodb://localhost:27017'
  }
  return 'mongodb+srv://todo-db-admin:D92dARWONO0t16uF@todo-cluster0-
ilc7v.mongodb.net/test?retryWrites=true'
}

const dbName = () => {
  const env = process.env.NODE_ENV
  if (env === 'test') {
    return 'todo-test'
  } else if (env === 'dev') {
    return 'todo-dev'
  }
  return 'todo-prod'
}

const apiRoot = ()  => {
  const env = process.env.NODE_ENV
  if (env === 'prod') {
    return ''
  }
  return 'https://api.klequis-todo.tk'
}

export default {
  mongoUrl: mongoUrl(),
  dbName: dbName(),
  apiRoot: apiRoot(),
  port: 3030
};
```

`db/dbFunctions.js` Changes

Move the /db directory from 'wrapping-calls-to-mongodb' to the current project
`dbFunctions.js` is using hard coded values for configuration. Change it to make use of /config.

```
import config from '../config'

...
// from
client = await MongoClient.connect(mongoUrl, { useNewUrlParser: true })
// to
client = await MongoClient.connect(config.mongoUrl, { useNewUrlParser: true })

...

// from
return { db: client.db(dbName) }
// to
return { db: client.db(config.dbName) }

...
```

Copy the test/ Directory

Move the /test directory from 'wrapping-calls-to-mongodb' to the current project

Add test and test-watch scripts to package.json, (copied from wrapping-calls-to-mongodb)

```
"test": "mocha --require @babel/register",
"test-watch": "export WATCH='watch' && nodemon --exec 'npm test'",
```

TODO: I don't understand `export WATCH='watch'`

Run Tests

Start MongoDB if it isn't already running

```
$ sudo service mongod start
```

Run the tests

```
npm run test-watch
```

Hopefully, all of the tests past. If not, debug them before moving on.

server/ Changes

Add

```
import morgan from 'morgan'
import todo from '../routes/todo-route'
import config from '../config'
```

Delete

```
const port = 3030
```

Delete

```
app.get('/', (req, res) => {
    res.status(200).send({ data: 'hello', error: '' })
})
```

Add

```
app.use(morgan('dev'))

app.use('/api/todo', todo)
app.get('/api', (req, res) => {
  console.error('Invalid endpoint!')
  res.send('Invalid endpoint!')
})


// Change `port` to `config.port`
app.listen(config.port, () => {
  console.log(`Events API server is listening on port ${config.port}`)
})

// ?
// export default app
```

Routes

TODO: add update route

TODO: currently findOneAndUpdate & objectIdFromHexString are not needed. findOneAndUpdate will be but I'm not sure about objectIdFromHexString

Create a directory named `routes`
Create a file named `todo-route.js`
Make the routes as below

TODO: explain routes and how to make them step by step

```
import express from 'express'
import { find, findById, insertOne, findOneAndDelete, findOneAndUpdate,
objectIdFromHexString } from '../db'
import { red, yellow } from '../logger'

const router = express.Router()

/*
    - assumes only { title: string } is sent
    - { completed: false } will be added to all new todos
```

```
 */

router.post('/', async (req, res) => {
  try {
    const td1 = req.body
    // yellow('post', td1)
    const td2 = {
      title: td1.title,
      completed: false,
    }
    const inserted = await insertOne(
      'todos',
      td2
    )
    res.send(inserted)
  } catch (e) {
    red('error', e)
    res.status(400).send(e)
  }
})

router.get('/', async (req, res) => {
  try {
    const todos = await find('todos')
    res.send(todos)
  } catch (e) {
    res.status(400).send(e)
  }
})

router.get('/:id', async (req, res) => {
  const id = req.params.id
  try {
    const todos = await findById('todos', id)
    res.send(todos)
  } catch (e) {
    res.status(400).send(e)
  }
})

router.delete('/:id', async (req, res) => {
  const id = req.params.id
  try {
    let todo = await findOneAndDelete('todos', id)
    if (!todo) {
      return res.status(404).send()
    }
    res.send(todo)
  } catch (e) {
    res.status(400).send()
  }
})
```

```
export default router
```

Testing Routes

use dbFunctions directly

You could use the http methods in `routes/todo-route.js` for test setup and teardown, but then there would be the issue of needing to use functions for the setup that have not been tested yet as noted when starting to write tests for `routes/todo-route.js`. To avoid this, us the functions in `db/dbFunctions.js` for setup and tear down.

avoiding two processes

If you were to wite a test and then run `npm run test`, the test would run but the server will be started twice. When the tests complete, one of the servers will still be done and the tests will not exit. While you can end them manually ctrl+c, this will present a problem when using continious integration as the test suite will not complete.

To fix this, make a change to `server/index.js`. Wrap `app.listen` in an if statement like this:

```
if (!module.parent) {
  app.listen(config.port, () => {
    console.log(`Events API is listening on port ${config.port}`)
  })
}
```

What this says is, if the server is launched directly, start the server. However, if the server is launched by another process, in this case Moca, don't start it. Now you will get only one instance of the server running and it will shut-down when the tests complete.

> One way to see this problem is to run `npm run test-watch`. With the `if` statement in place Nodemon will print out:

```
[nodemon] clean exit - waiting for changes before restart
```

> Without the `if` statement, this will not be printed out.

decrease clutter

To decrease clutter, disable the dbFunctions tests by adding `.skip` to the outermost `describe()`:

dbFunctions.test.js

```
describe.skip('dbFunctions', () => {

  ...
```

~~~
})

```# React Client Part II ## ZZ Home

Welcome to the react-typescript-todo-ex wiki!



# Next
- implement Express server
- fix: change to babel.config.js?
  - currently the master project both babel.config.js and .babelrc
- ecosystem.config.js
- todo-route.js
- fix: does master project need env/env-vars.js?
- db/*
- config/*






# Outline


## Home
- Home.md - this outline

## Introduction
- Introduction.md
- Development-Machine-Setup.md

## Create the Express Server
- Server-Config.md
- Create-a-MongoDB-Database.md
- Server Project Setup
  - New folder
  - npm -init -y
  - .gitignore
  - .babelrc
  - logger
- Wrapping-Calls-to-MongoDB.md
  -
  - Tests

## Create the client

## Deploy Serer to DigitalOcean
- Server setup
  - NPM
~~~

```
  - PM2
  - Nginx Reverse Proxy

## Deploy Client to AWS S3
- Create the bucket
- Upload files

## DNS Setup

## Setup CloudFront and HTTPS
- Get a certificate for your domain
- Create a CloudFront distribution using HTTPS
Carl Becker (klequis) https://carlbecker.com# ZZ Along the Way

- Should read this: [The Little MongoDB Schema Design Book]
(http://learnmongodbthehardway.com/schema/)

- Source doc for .mjs: [ECMAScript Modules]
(https://nodejs.org/api/esm.html#esm_ecmascript_modules)

- Utility for converting string times as miliseconds and the other way around:
[ms](https://github.com/zeit/ms)

- A 'small debugging utility' with 35 million weekly downloads!: [debug]
(https://www.npmjs.com/package/debug)

- [UFW Essentials: Common Firewall Rules and Commands]
(https://www.digitalocean.com/community/tutorials/ufw-essentials-common-
firewall-rules-and-commands)

# ZZ Setup Domain

Log into the registrar that manages you DNS and change the name servers to
ns2.digitalocean.com
ns2.digitalocean.com
ns2.digitalocean.com

This change can take up to 48 hours, but in my experience it is one to several
hours. Your mileage my vary.

Back in DigitalOcean, make sure you have the currect team selected and then
click *Networking* in the left hand navigation.

In the Domains tab, enter the name of your domain (e.g., todo.tk) then click
*Add Domain*. This should take you to the page for adding records to the
domain. If not, click directly on your new domain.

Add 3 records. For each, enter a host name and choose your server under *WILL
DIRECT TO*.

| HOSTNAME | WILL DIRECT TO |
| -------- | -------------- |
```

~~~
| * | todo-server |
| @ | todo-server |
| www | todo-server |

* Note that in each case the domain name is added. You do not need to type it
in.

> TODO: confirm need to have the '*' record as droneevents.live does not









# Creating Test Express Server
Next we will create a simple Express server to use for testing our machine
build and working with the additional steps of setting-up PM2 & Nginx as a
reverse proxy server

```js
$ cd
$ mkdir todo-server
$ cd todo-server
$ npm init -y
~~~

The response will be as belo. We will discuss changeing these values later.

```
Wrote to /home/doadmin/todo-server/package.json:

{
  "name": "todo-server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

```
$ npm i express
$ mkdir server
$ touch server/index.js
$ nano server/index.js
```

Contents of the file

```
const express = require('express')

const port = 3030
const app = express()

app.get('/', (req, res) => {
  res.send('Response from todo-server')
})

app.listen(port, () => {
  console.log(`Events API server is listening on port ${port}`)
})
```

Run the server

```
$ node server
```

Output

```
Events API server is listening on port 3030
```

To test the server, open another terminal on your server and use 'curl' to call it

```
curl http://localhost:3030
```

In the console you should see

```
Response from todo-server
```