Assignment notes

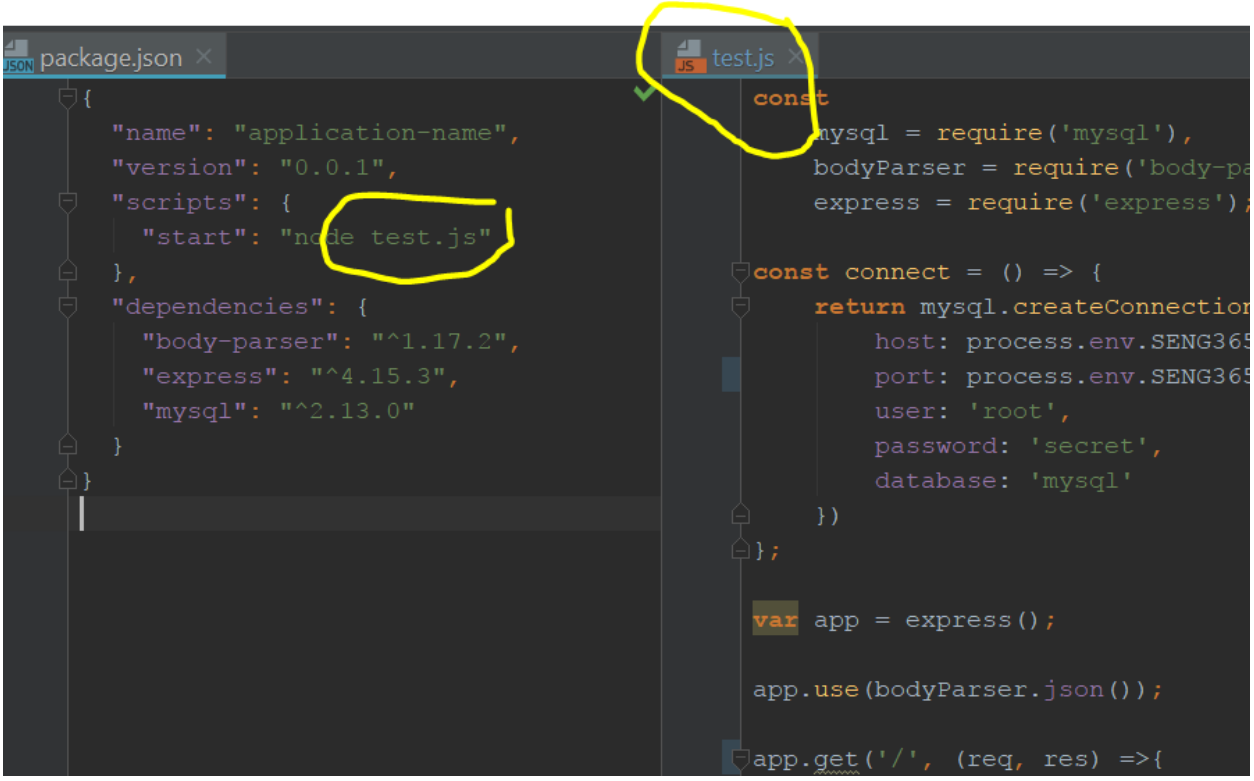
Troubleshooting the standard deployment to csse-s365

If you haven't made your own changes to the deployment files in your repo, these checks may help if/when things don't work.

Can't connect to your app

1. *Is the deployment process trying to run the right program?*

   If you expect to run your application by `"node test.js"`, then the start script line in `package.json` must match: `"start": "node test.js"`

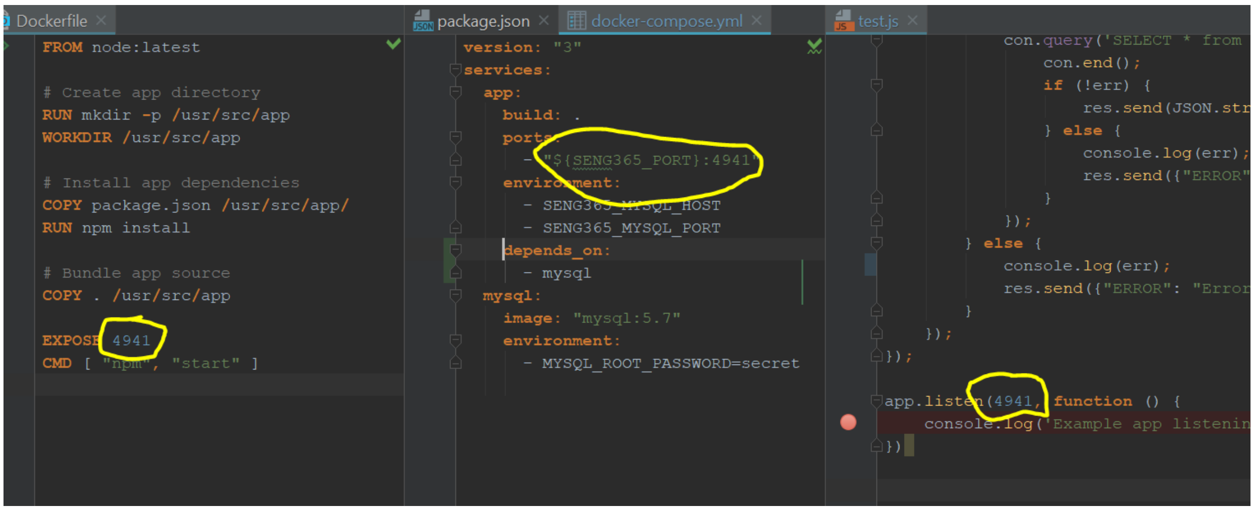


```
{
  "name": "application-name",
  "version": "0.0.1",
  "scripts": {
    "start": "node test.js"
  },
  "dependencies": {
    "body-parser": "^1.17.2",
    "express": "^4.15.3",
    "mysql": "^2.13.0"
  }
}
```

```
const
    mysql = require('mysql'),
    bodyParser = require('body-pa
    express = require('express');

const connect = () => {
    return mysql.createConnectio
        host: process.env.SENG365
        port: process.env.SENG365
        user: 'root',
        password: 'secret',
        database: 'mysql'
    })
};

var app = express();

app.use(bodyParser.json());

app.get('/', (req, res) =>{
```

2. *Is the application exposed on the expected port?*

   The port number in "app.listen()" (assuming Express) must match the port number EXPOSEd in the Dockerfile and the port number under `ports` in docker-compose.yml. By convention these should all be 4941. Note that this isn't the same port as your assigned SENG365_PORT (that's mapped in the circled line in the middle of the image below.)



Dockerfile

```
FROM node:latest

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 4941
CMD [ "npm", "start" ]
```

docker-compose.yml

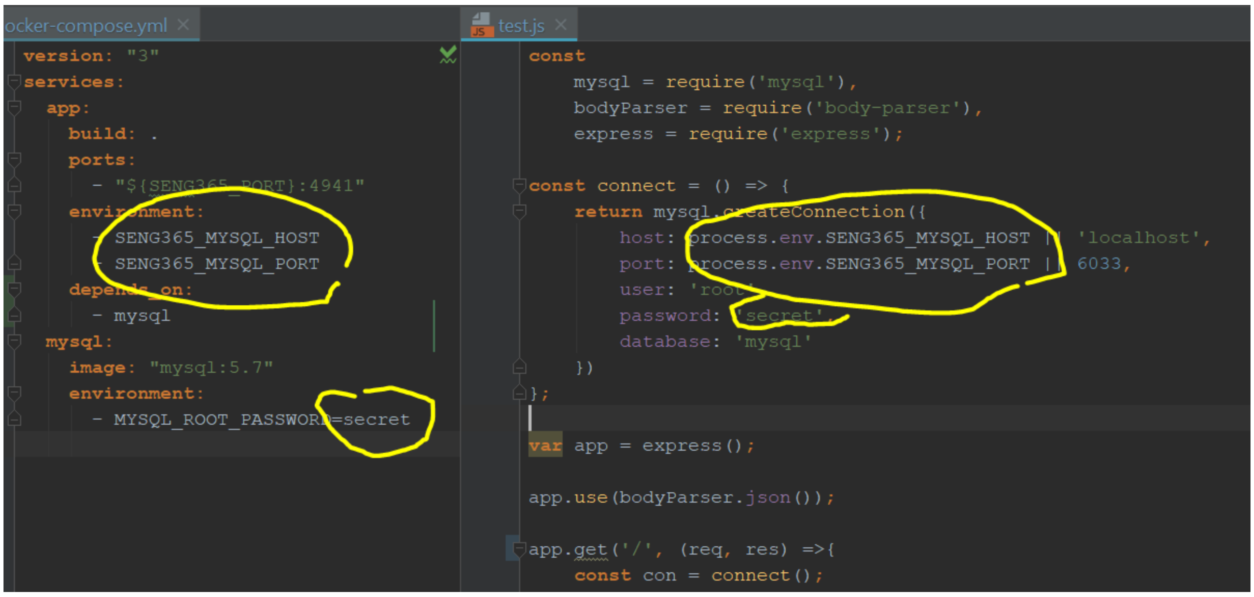
```
version: "3"
services:
  app:
    build: .
    ports:
      - "${SENG365_PORT}:4941"
    environment:
      - SENG365_MYSQL_HOST
      - SENG365_MYSQL_PORT
    depends_on:
      - mysql
  mysql:
    image: "mysql:5.7"
    environment:
      - MYSQL_ROOT_PASSWORD=secret
```

test.js

```
            con.query('SELECT * from
                con.end();
                if (!err) {
                    res.send(JSON.str
                } else {
                    console.log(err);
                    res.send({"ERROR"
                }
            });
        } else {
            console.log(err);
            res.send({"ERROR": "Error
        }
    });
});

app.listen(4941, function () {
    console.log('Example app listenin
})
```

Can't connect to MySQL from your app

1. The connection must have user `root`, password `secret`, host `SENG365_MYSQL_HOST` and port `SENG365_MYSQL_PORT`.



ocker-compose.yml

```
version: "3"
services:
  app:
    build: .
    ports:
      - "${SENG365_PORT}:4941"
    environment:
      - SENG365_MYSQL_HOST
      - SENG365_MYSQL_PORT
    depends_on:
      - mysql
  mysql:
    image: "mysql:5.7"
    environment:
      - MYSQL_ROOT_PASSWORD=secret
```

test.js

```
const
    mysql = require('mysql'),
    bodyParser = require('body-parser'),
    express = require('express');

const connect = () => {
    return mysql.createConnection({
        host: process.env.SENG365_MYSQL_HOST || 'localhost',
        port: process.env.SENG365_MYSQL_PORT || 6033,
        user: 'root',
        password: 'secret',
        database: 'mysql'
    })
};

var app = express();

app.use(bodyParser.json());

app.get('/', (req, res) =>{
    const con = connect();
```

For reference, `SENG365_MYSQL_HOST` is set for you to `mysql`, the service name in `docker-compose.yml`, and `SENG365_MYSQL_PORT` is set to 3306, the standard MySQL port.

Authentication using Express middleware

Token-based authentication is a simple solution for creating endpoints that require a user to be identified, whilst still maintaining a stateless server.

The general steps behind token-based authentication are as follows:

1. User supplies username and password.
2. Server checks credentials.
3. A unique token is returned to the client.
4. The token is provided with each further request by the client on behalf of the user.
5. The server checks that requests to protected endpoints contain a valid token.

For Assignment 1 and 2, the token in steps 3-5 will be passed in a `X-Authorization` header field.

What does a token look like?

A token is just a unique string of characters, often base64, for example: `dBjftJeZ4CVP-mB92K`, although for our assignments you can use any string you like. This reduces the complexity required to get a simple authentication system up and running (although clearly this is a bad idea for a real project!)

How can I implement token authentication?

Adding token authentication to an API is easy with Express middleware. Middleware is software that runs between two different parts of the software, sometimes described as software glue. In fact, you've already seen an example in Lab 2 with `bodyParser`.

Express middleware functions are just functions that take three specific parameters: `req`, `res` and `next`. The first two parameters we've encountered before, while `next` is a callback for the next step in the pipeline. The Express documentation has some helpful information on writing middleware.

Here's an example of middleware for the assignments, where `isValidToken` is a function you'd have to write to check that the token in X-Authorization is valid:

```
const myMiddleware = (req, res, next) => {
    if (isValidToken(req.get('X-Authorization'))) {
        next(); // if we have a valid token, we can proceed
    } else {
        res.sendStatus(401); // otherwise respond with 401 unauthorized
    }
}
```

Middleware in Express can be applied globally, or per-endpoint. If only some endpoints are to be authenticated, include your middleware function as a parameter to each protected endpoint, for example:

```
app.get('/path', myMiddleware, callback);
app.post('/path', myMiddleware, callback2);
```

The function `myMiddleware` will run before the `callback` (or `callback2`) function is called.

On the other hand, if the middleware should be applied to every endpoint, then include this line before your other routes:

```
app.use(myMiddleware);
```

Production vs Development differences for MySQL

Because we're using a continuous deployment pipeline, you need to take into account that you'll be dealing with (at least) two different runtime environments for your application:

1. Your *development* system, for example one of the lab machines. If you use the MySQL VM we've provided (see *Lab 3: Persisting data in Node*), your application will connect to MySQL on `localhost`,

port 6033 (chosen to prevent clashes with other MySQL instances.)
2. The *production* deployment on `csse-s365.canterbury.ac.nz`. MySQL in production is deployed within a docker container, and consequently your app will connect to MySQL on `mysql`, port 3306 (the standard MySQL port).

! You'll need to account for this difference in configuration in your application. If you don't, your development environment will work well, but things *will* break in production (and we grade on the production system.)

We have set two environment variables in production, `SENG365_MYSQL_HOST` and `SENG365_MYSQL_PORT`, that will help. For example, instead of:

```
const con = mysql.createConnection({
  host: 'localhost',
  port: 6033,
  user: 'root',
  password: 'secret',
  database: 'mysql'
});
```

use:

```
const con = mysql.createConnection({
  host: process.env.SENG365_MYSQL_HOST || 'localhost',
  port: process.env.SENG365_MYSQL_PORT || 6033,
  user: 'root',
  password: 'secret',
  database: 'mysql'
});
```

If this seems like magic, things will be clearer after you have completed Lab 3.

Where does my application run when deployed?

When you push your code to your `seng365-2017/<usercode>` git repository on eng-git, the continuous deployment process runs it in docker containers on the shared VM at `csse-s365.canterbury.ac.nz`. We map the port you use to access your app in development (we expect your app is listening on port 4941) to a unique port on the VM associated with your usercode. That port can be found in your project settings on eng-git under "variables" as `SENG365_PORT`.

So to connect to your application in production use the url:

`http:\\csse-s365.canterbury.ac.nz:<SENG365_PORT>`

Note that this port mapping happens automatically on deployment; you should **not** change the port you listen on in your application. For the port mapping defined in the `docker-compose.yml` file, it's important that your app listens on port 4941.

MySQL will be cleared each deploy in Production

Currently we clear out your MySQL DB on `csse-s365.canterbury.ac.nz` each time you do a deploy (that is, each time you push to eng-git.) This is the result of the `docker-compose down` line in the `.gitlab-ci.yml` in the root directory of your project.

Therefore your app should create your database schema itself on first run (and add any sample data you might have.) This isn't especially production-like but it is convenient all round.

App must wait for MySQL in Production

Because your app and the MySQL instance are started simultaneously, it's not safe to expect that MySQL will be ready when your app attempts to first connect.

In the latest docker-compose.yml file, the application is not started until MySQL has started (the `depends_on` block in the YAML)

The correct thing to do is to retry the connection (after a short wait) if it is initially refused.

How does this continuous deployment *work*?

1. You hackity hack...
2. You commit your changes to your git repo on `eng-git.canterbury.ac.nz`.
3. That triggers the GitLab CI runner, which starts a GitLab docker executor on the build VM (as defined in `/etc/gitlab-runner/config.toml`) to build your application within a docker container.
4. The executor runs the build defined in your `.gitlab-ci.yml`, by passing each script line in the build section to a container which includes binaries for `docker compose` in an otherwise standard docker-in-docker image. (We use a form of the "docker socket binding" method)
5. The docker executor now runs the build script in `.gitlab-ci.yml` (`docker compose build`) in the build container.
6. `Docker compose build` follows the build steps from `docker-compose.yml` to build the application.
7. If the build step succeeds, the CI runner on the VM then runs the scripts in the deploy section of `.gitlab-ci.yml` in the same way.
8. This calls `docker compose down` to stop any previous version of your app, and then `docker compose up` to start the now-built services defined in `docker-compose.yml`, using that files definition for the port mapping from container to host ports.
9. Your application is now up-and-running.
10. As `docker compose up` in `.gitlab-ci.yml` has the `-d` flag, the CM runner doesn't wait for it to finish and so can immediately mark the deploy as complete.