NodeJS SQL Setup in Azure

Most applications use a database to store information. For your web application you may choose to use in-memory variables, local files, or a database. If you choose to use a database you will need to configure your Azure instance to run the database software. The easiest way, for the Microsoft Imagine subscription, is to use Microsoft SQL Server (MsSQL Server) hosted on Azure.

1 Creating the SQL Database and Server

Navigate to your Azure portal site and click on the **SQL Databases** link in the sidebar and then **Add** at the top of the webpage. We're going to start filling in the details for our Microsoft SQL Server Database. First make sure the **subscription** selected says **Azure for Students** and the **resource group** is the same as the WebApp you created in previous labs. Give your database a unique name, I called mine *mydb* as I'm not imaginative enough.

For the **server** option, click **Create new** below the input field. You should see a sidebar to the right appear. Fill it in as shown in fig. 1, but with your own details. **Make sure to remember the login details!** Click select in the right sidebar and then on **Configure database** at the bottom of the middle window.

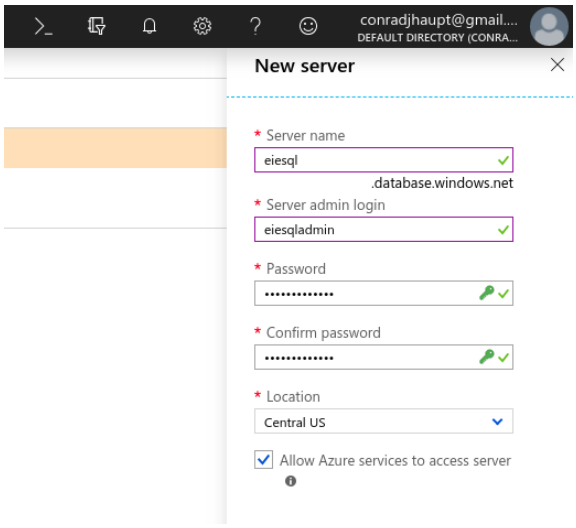

Figure 1: Creating a new Microsoft SQL Server on Azure

In the new window that appears, select **Basic** as the database plan. If you feel like you need a different plan, then do so but be aware that your **Azure for Students** subscription has limited credit. Leave the **Data max size** at $2GB$ and click **Apply**.

We are now ready to create our database. Click **Review + Create** at the bottom of the webpage and then **Create**. Your SQL database will take some time to deploy properly.

1.1 SQL Server Firewall Rule

To access our SQL database, not only do we need an internet connection to Azure but the SQL Server's firewall needs to be modified so we can connect to it while testing on our own machines. To do this, type **SQL server** into the Azure Portal search bar at the top of the page and then click on the **SQL server** option that appears. You should now see a list of your servers including the one we created early. Click on its name to get to the overview page. Your webpage should look something like that shown in fig. 2.

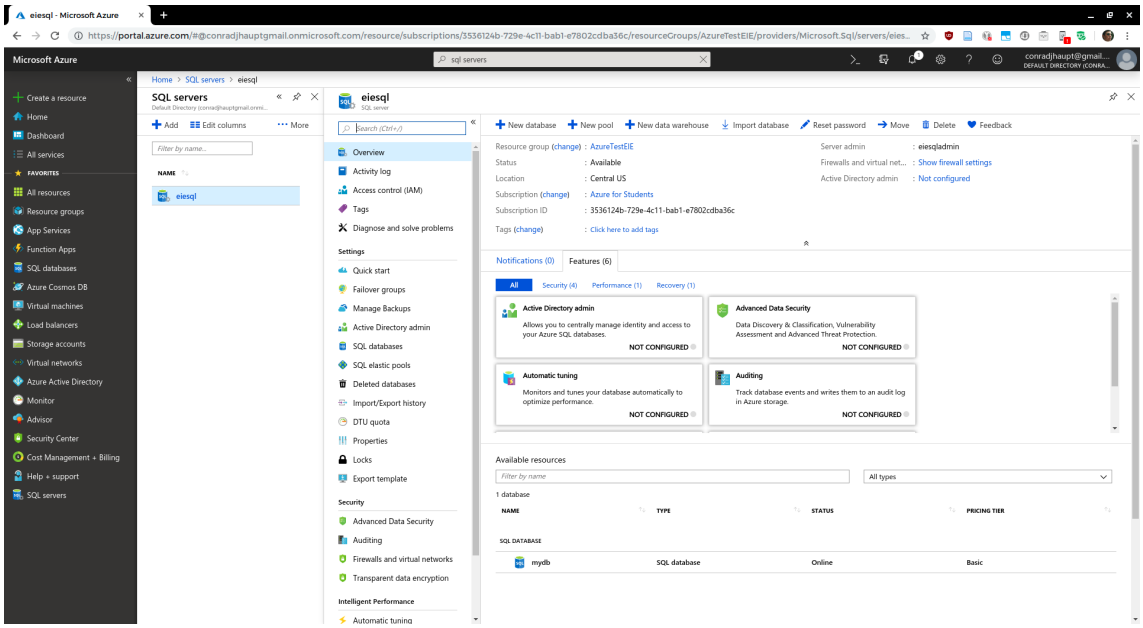

Figure 2: Azure SQL Server Overview Page

Navigate to the **Firewalls and Virtual Networks** section using the SQL Server sidebar, input the following details into the IP-Address input fields, and then click **Save** at the top of the page.

RULE NAME Wits network

START IP *146.141.0.0*

END IP *146.141.255.255*

Your window should look like that shown in fig. 3. The reason we do this is so the SQL Server will accept connections from any computer on the Wits network. If you are working off campus, you will need to either add your IP address to the firewall or VPN in to the Wits network remotely. For production databases, the IP addresses are fixed to internal subnets so that no-one from outside the data-centre can access the database. The frontend server would then have two IP addresses, the one the public accesses on and an internal one through which it connects to the database server.

If you want to run SQL queries on the database, navigate to the SQL Database's page using the Portal Sidebar then click on **Query Editor (preview)** in the database's sidebar. Login using the credentials you defined earlier. The **query window** that appears allows you to run standard SQL queries on your database without running any code. This is an easy place to define your **schema**.

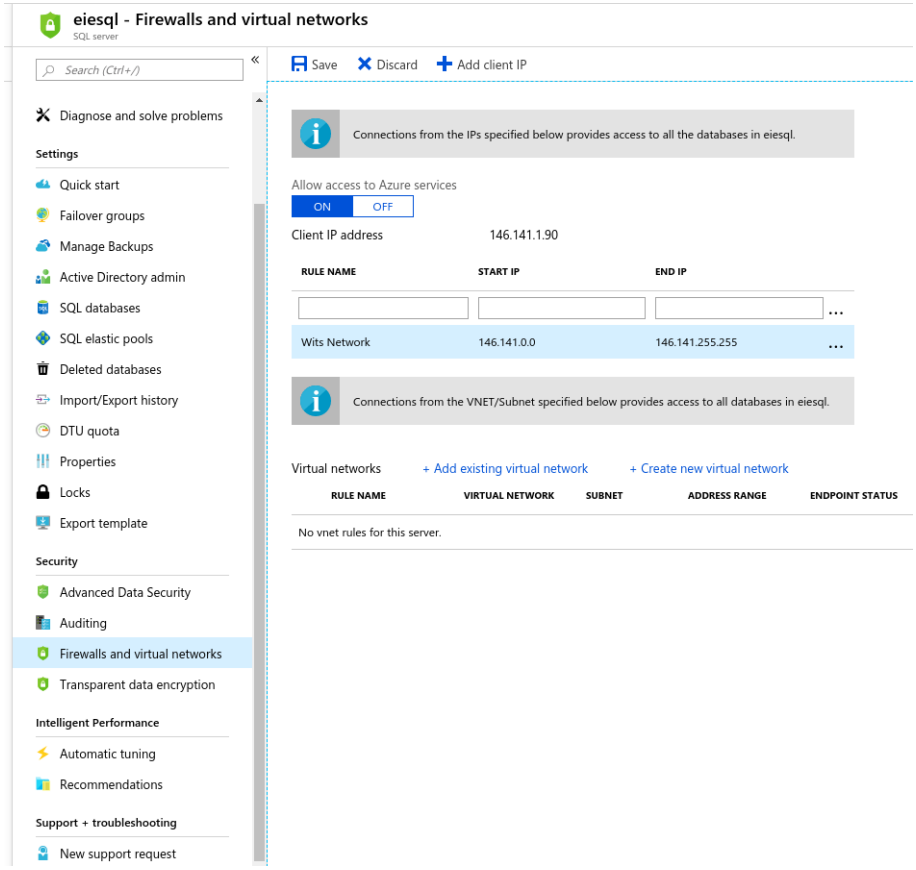

Figure 3: Azure SQL Server Firewall Rules for Wits Network

2 Connecting NodeJS and MsSQL

To connect our NodeJS WebApp to our SQL server we need a new NPM package. Run `npm install --save-exact mssql` in your NodeJS project folder to install it. We are going to create a **database module** for our application to manage the connection to our SQL server. Create a file called **db.js** and add the following code to **db.js**.

```
let mssql = require("mssql")

// Make sure this is private to this module
let config = {
  server: "<your SQL server name>.database.windows.net",
  database: "<your database name>",
  // Put login details in env. variables for security
  user: "<your username>",
  password: "<your password>",
  port: 1433,
  // Required for Azure
  options: {
    encrypt: true
  },
  pool: {
    max: 10,
    min: 0,
```

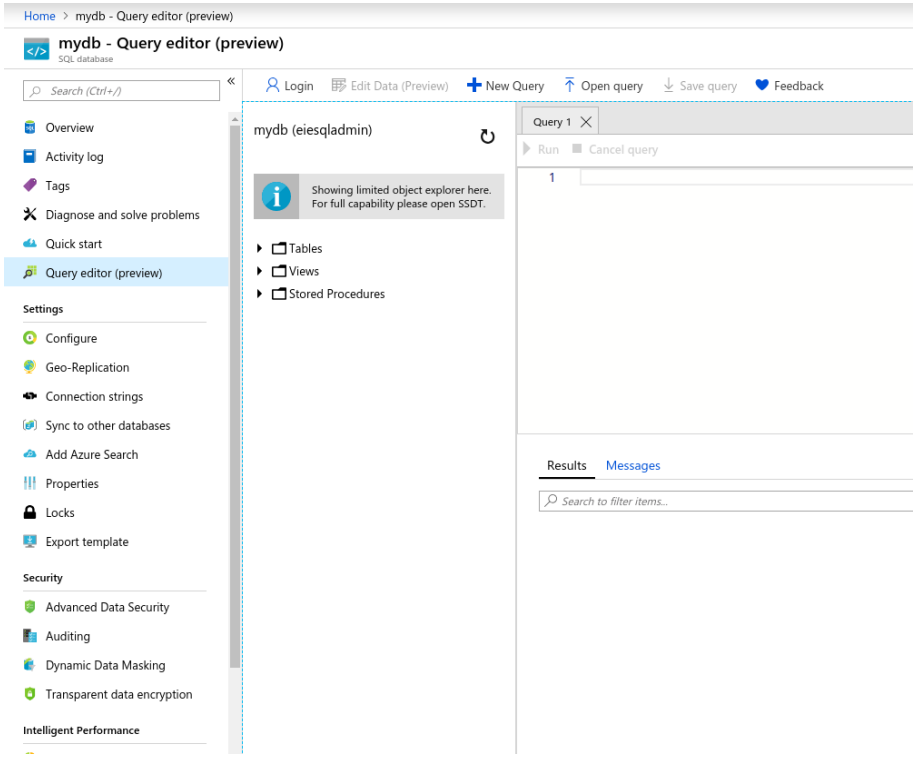

Figure 4: Azure SQL Database Query Editor

```
    idleTimeoutMillis: 30000
  }
}

// Get a mssql connection instance
let isConnected = true
let connectionError = null
let pools = new mssql.ConnectionPool(config)
  .connect()
  .then(pool => {
    console.log('Connected to DB')
    return pool
  })
  .catch(err => {
    // Handle errors
    isConnected = false
    connectionError = err
    console.log(err)
  })

module.exports = {
  sql: mssql,
  pools: pools,
  isConnected: isConnected,
  connectionError: connectionError
}
```

You can get the details for the **config** variable from the Azure portal page for the database

we created. See fig. 5.

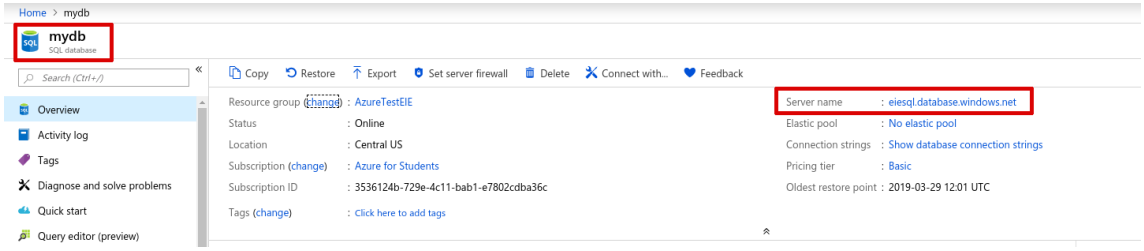

Figure 5: Where to get your SQL Database name and URL

NOTE: For security reasons, you **should not commit your login details to git**. Use environment variables instead.

To store our login details in environment variables, navigate to your Azure portal, WebApp page, Configuration settings, and then `Application Settings`. Here you can create any environment variable you want and it will be accessible by your NodeJS application. This is shown in fig. 6.

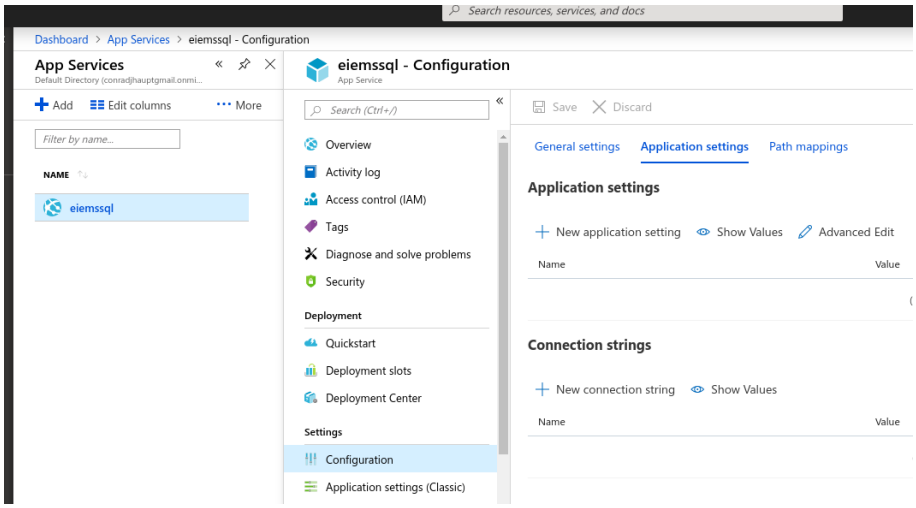

Figure 6: Environment Variable Settings for WebApp

Click on `New Application String` and input the variable name and value you want. For example, you could put in the username for your SQL server and database as shown in fig. 7. Click okay and add more variables if you need to.

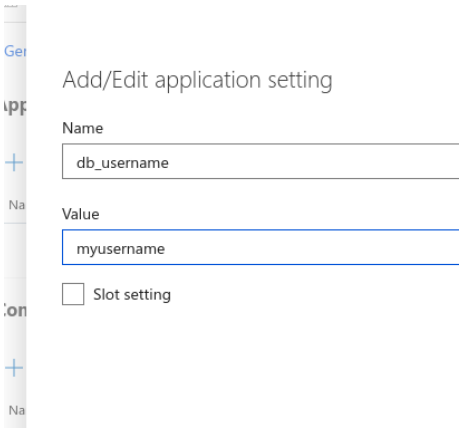

Figure 7: Creating a new Environment Variable

Once you are done, click save. These variables should now be accessible in your NodeJS application using `process.env.<variable name>`. To have the same variables accessible on your machine, you should search how to create the same environment variables on the operating system you use.

Now add a route to your NodeJS app as shown below.

```
let db = require('./db.js')
...
mainRouter.get('/database', function (req, res) {
  // Make a query to the database
  db.pools
    // Run query
    .then((pool) => {
      return pool.request()
        // This is only a test query, change it to whatever you need
        .query('SELECT 1')
    })
    // Send back the result
    .then(result => {
      res.send(result)
    })
    // If there's an error, return that with some description
    .catch(err => {
      res.send({
        Error: err
      })
    })
})
```

Run your code **locally** and test the **/database** path in your browser. You should see a response like this:
`{"recordsets":[[{"":1}]],"recordset":[{"":1}],"output":{},"rowsAffected":[1]}`.

You do not have to send the SQL query response to the user, instead you can process it as JSON or Javascript objects. To find out more about the **mssql** library, navigate to https://www.npmjs.com/package/mssql.