CAB230 Web Computing
TLS and Localhost

In the last prac we extended the World Cities API app to include some important middleware and to include API docs. In this prac we are going to walk through HTTPS 'deployment' of the server on localhost using a self-signed certificate. Working through this example will also give us a good opportunity to take another look at the structure of an Express application as generated using Express generator.

Much of what follows draws upon a particular Stack Overflow question, though other sources have been consulted and some additional information has been included: https://stackoverflow.com/questions/21397809/create-a-trusted-self-signed-ssl-cert-for-localhost-for-use-with-express-node .

TLS is covered in the week 8 lecture, but the most important aspect here is that we need to generate a certificate that complies with the X.509 standard. Such certificates include an identity and a public key. See https://en.wikipedia.org/wiki/X.509 for more details. Certificates come in different shapes and sizes and a good guide can be found here: https://www.cloudflare.com/learning/ssl/types-of-ssl-certificates/.

The Certificate Request:
Our job initially is to create a certificate signing request, which requires a configuration file like the one below, which is supplied in the starter-files as `req.cnf`. You should make a new directory under the main application level and call it `sslcert`, and copy this file into the new directory. We will make any changes we feel are necessary and then create the key and certificate files necessary to use TLS.

The main purpose of the fields shown below is to specify the organisation identity and the purpose of the site for usage of the key. You should not modify anything above `[req_distinguished_name]`. The relevant fields are then:

- `C – Country Code (2 Letter)`
- `ST – State or province. Not abbreviated.`
- `L – Location. Not abbreviated.`
- `O – Organisation. Not abbreviated.`
- `OU – Organisational Unit. Not abbreviated.`
- `CN – Common Name – the host protected by the certificate.`

In the CSR doc I have used some obvious choices for these fields, with the fictitious www.localhost.com as the `Common Name`. You should leave the extensions unchanged, and perhaps choose an organisational unit to suit yourselves. The main goal is to then create a set of credentials that can then be used in the creation of an HTTPS server. We will then register the certificate with the installed OS so that it is treated (on the present machine) as though it had been supplied by a trusted Certificate Authority. The version of the config file I used may be found at the top of the next page:

```
[req]
distinguished_name = req_distinguished_name
x509_extensions = v3_req
prompt = no
[req_distinguished_name]
C = AU
ST = Queensland
L = Brisbane
O = Queensland University of Technology
OU = CAB230 Class
CN = www.localhost.com

[v3_req]
keyUsage = critical, digitalSignature, keyAgreement
extendedKeyUsage = serverAuth
subjectAltName = @alt_names
[alt_names]
DNS.1 = www.localhost.com
DNS.2 = localhost.com
DNS.3 = localhost
```

The next step is to use the `openssl` tool to generate the key and certificate. If the program is not available on your machine, use the installation process for your operating system to make it available. For linux, just use your package manager. For windows, `openssl` ships with the git bash shell at `/mingw64/bin/openssl` or via the Windows Subsystem for Linux, where it is already installed in the Ubuntu 18.04 image. Those uncomfortable with this approach can look at the binaries available here: https://wiki.openssl.org/index.php/Binaries.

The choices for https://slproweb.com/products/Win32OpenSSL.html are given here:

File	**Type**	**Description**
Win64 OpenSSL v1.1.1b Light EXE | MSI (experimental)	3MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v1.1.1b (Recommended for users by the creators of OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to local and state laws. More information can be found in the legal agreement of the installation.

Note that by default the windows installer may not add the location to the path, and so you may need to include it manually. I also had some difficulties with the MSI and relied on the EXE, which worked fine.

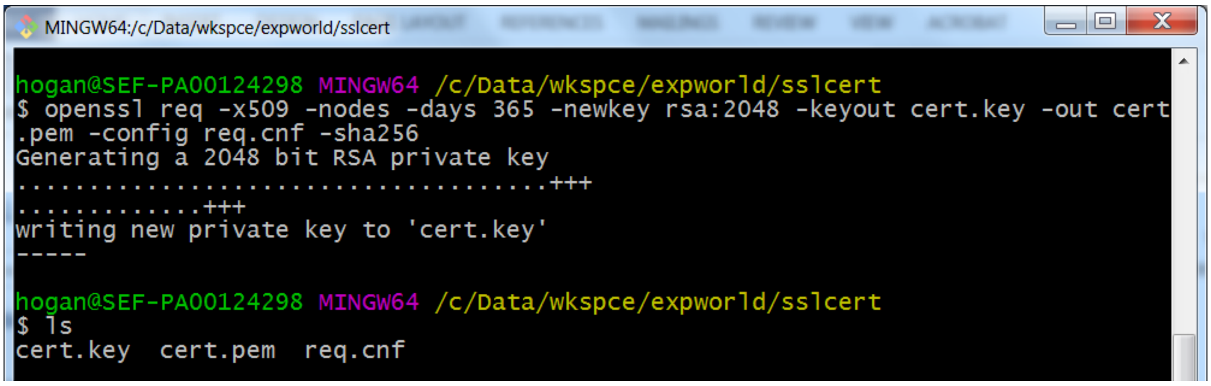
Once you have finished any edits to the file, you should execute the following command:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout cert.key -out cert.pem -config req.cnf -sha256
```

Here `req` is the overall command for certificate management, with the options:

- `-x509 – compliance with X509`
- `-nodes – private key is not encrypted (most likely no-des rather than nodes)`
- `-days 365 – the valid time range for the certificate.`
- `-newkey – generate a private key and certificate`

The remainder of the options concern input of the config file and choices of ciphers. Full details are available here: https://www.openssl.org/docs/man1.0.2/man1/req.html. You may also look at https://www.sslshopper.com/article-most-common-openssl-commands.html.



```
hogan@SEF-PA00124298 MINGW64 /c/Data/wkspce/expworld/sslcert
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout cert.key -out cert
.pem -config req.cnf -sha256
Generating a 2048 bit RSA private key
.....................................+++
.............+++
writing new private key to 'cert.key'
-----

hogan@SEF-PA00124298 MINGW64 /c/Data/wkspce/expworld/sslcert
$ ls
cert.key  cert.pem  req.cnf
```

The screenshot above shows the result and the credentials we will use with the app.

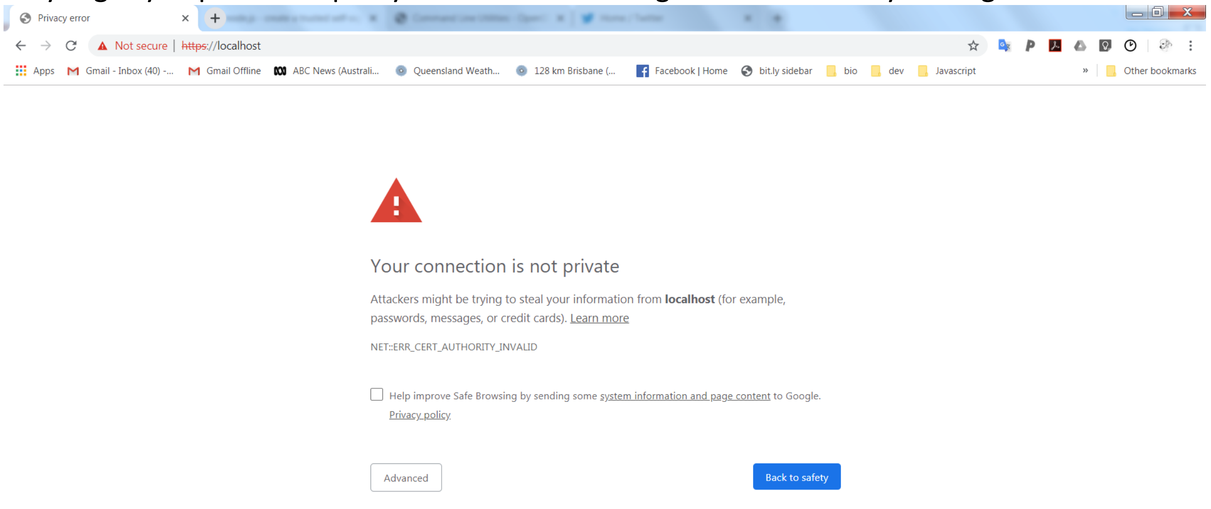
The Express app and HTTPS:
Including the credentials requires that we first load them from disk. This is best done using synchronous calls as shown. As you will recall, `fs` is the filesystem module:

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
const swaggerUI = require('swagger-ui-express');
const swaggerDocument = require('./docs/swagger.json');

const fs = require('fs');
const https = require('https');
const privateKey = fs.readFileSync('./sslcert/cert.key','utf8');
const certificate = fs.readFileSync('./sslcert/cert.pem','utf8');
const credentials = {
  key: privateKey,
  cert: certificate
};

var app = express();
```

The credentials object is established and we can go ahead and create the server using the app and these credentials. As is customary with HTTPS, we serve the responses on port 443. So far, everything has gone according to plan, and we expect a secure server. If you are running the server on a linux machine you will need to ensure that it is started with root level privileges. There is a standard gotcha in which you run the server and it tells you that it is working and then it quickly falls over with an error.

```
const server = https.createServer(credentials,app);
server.listen(443);

module.exports = app;
```

When we type https://localhost into the browser we see the following result, though this may vary slightly dependent upon your browser and organisation security settings:



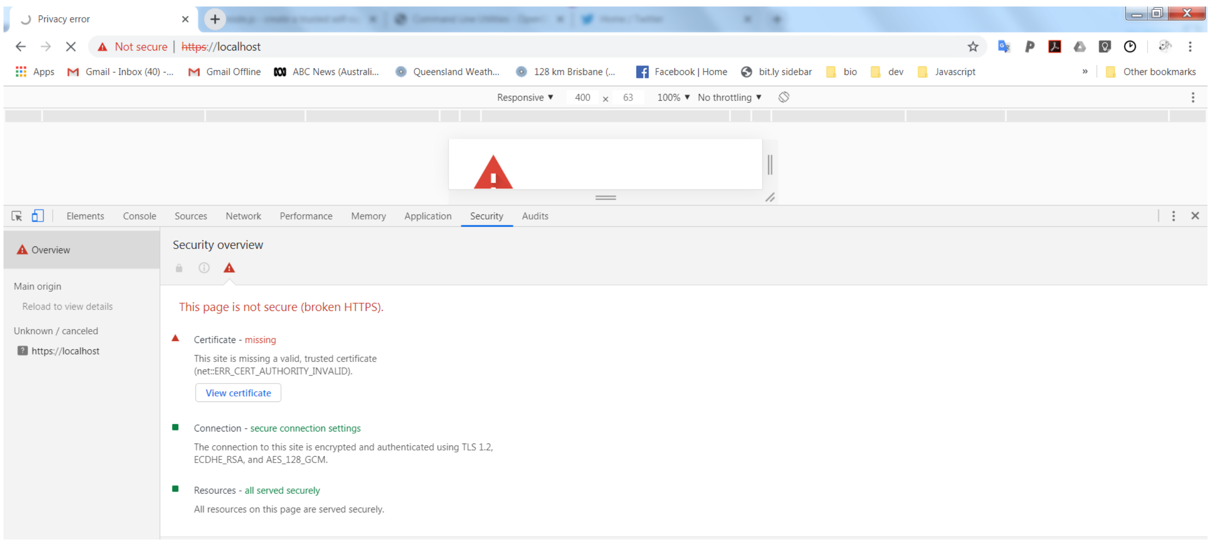
Your connection is not private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

Help improve Safe Browsing by sending some system information and page content to Google. Privacy policy

The next step is to inspect the result, and we select immediately the security tab in devtools:



Security overview

This page is not secure (broken HTTPS).

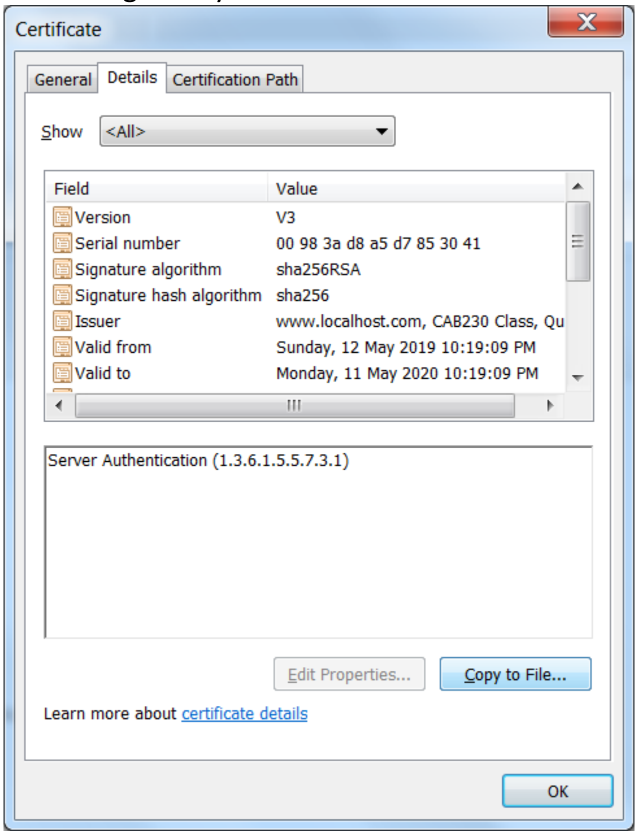
Certificate - missing
This site is missing a valid, trusted certificate (net::ERR_CERT_AUTHORITY_INVALID).

Connection - secure connection settings
The connection to this site is encrypted and authenticated using TLS 1.2, ECDHE_RSA, and AES_128_GCM.

Resources - all served securely
All resources on this page are served securely.

While it may be hard to see in the screenshot, the issue is reported as an unsecured page, due to broken HTTPS. The obvious next step is to accept the invitation to view the certificate.

As we can see, the certificate is not trusted. We see the validity range and a fair bit of information, but still, we don't get very far. Select the details tab on the Certificate viewer:

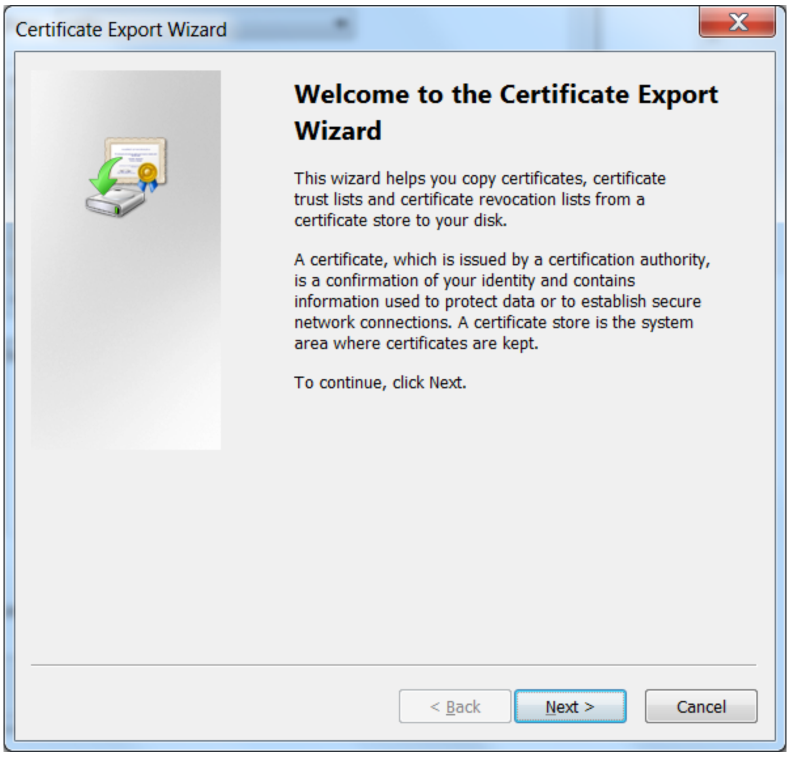


Field	Value
Version	V3
Serial number	00 98 3a d8 a5 d7 85 30 41
Signature algorithm	sha256RSA
Signature hash algorithm	sha256
Issuer	www.localhost.com, CAB230 Class, Qu
Valid from	Sunday, 12 May 2019 10:19:09 PM
Valid to	Monday, 11 May 2020 10:19:09 PM

Server Authentication (1.3.6.1.5.5.7.3.1)

Learn more about certificate details

We can see how the fields have been processed, and we now export the certificate to file using the export button that we can see at the bottom of the image. The wizard appears and we follow the prompts.



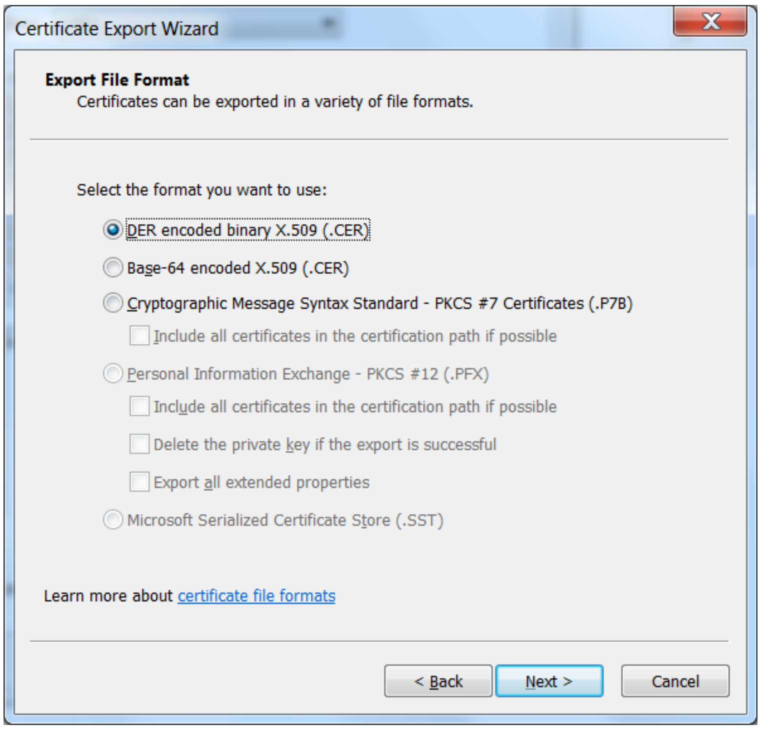
Welcome to the Certificate Export Wizard

This wizard helps you copy certificates, certificate trust lists and certificate revocation lists from a certificate store to your disk.

A certificate, which is issued by a certification authority, is a confirmation of your identity and contains information used to protect data or to establish secure network connections. A certificate store is the system area where certificates are kept.

To continue, click Next.

We accept the default format and export the file to some safe directory. Here I used the filename localhost.cer.


Certificate Export Wizard

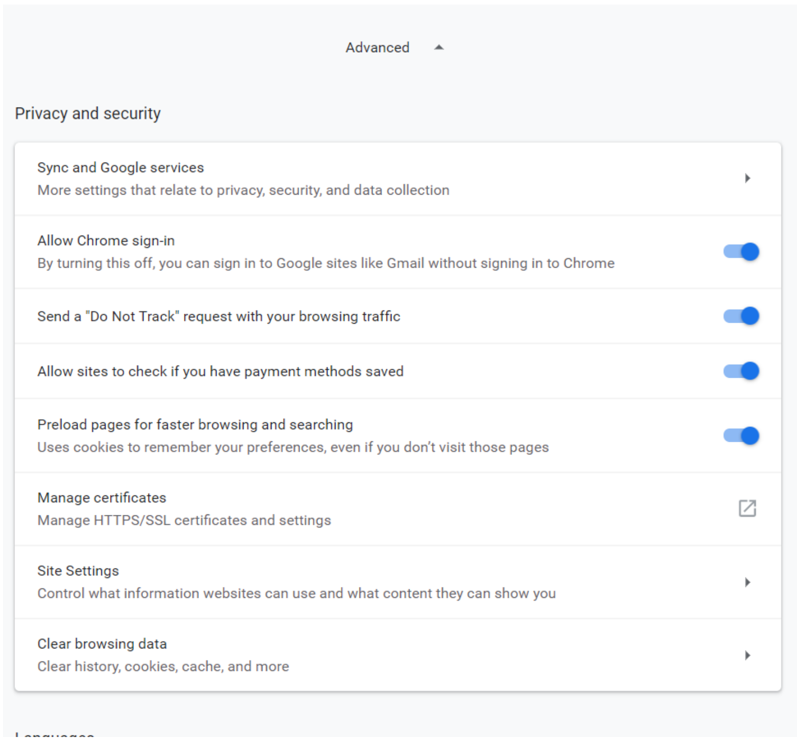
Export File Format
Certificates can be exported in a variety of file formats.

Select the format you want to use:

[✓ DER encoded binary X.509 (.CER)]
Base-64 encoded X.509 (.CER)
Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)
Include all certificates in the certification path if possible
Personal Information Exchange - PKCS #12 (.PFX)
Include all certificates in the certification path if possible
Delete the private key if the export is successful
Export all extended properties
Microsoft Serialized Certificate Store (.SST)

Learn more about certificate file formats

Here we work only with Chrome, though the process is similar (and also different ☺) for some other browsers. Type chrome://settings/ into the address bar, and then you will see the usual settings page. Scroll down and select 'Advanced' and eventually you will see the options below. By clicking at the right of 'Manage certificates', we see enter the certificate dialogue on the next page.


Advanced

Privacy and security

Sync and Google services
More settings that relate to privacy, security, and data collection

Allow Chrome sign-in
By turning this off, you can sign in to Google sites like Gmail without signing in to Chrome

Send a "Do Not Track" request with your browsing traffic

Allow sites to check if you have payment methods saved

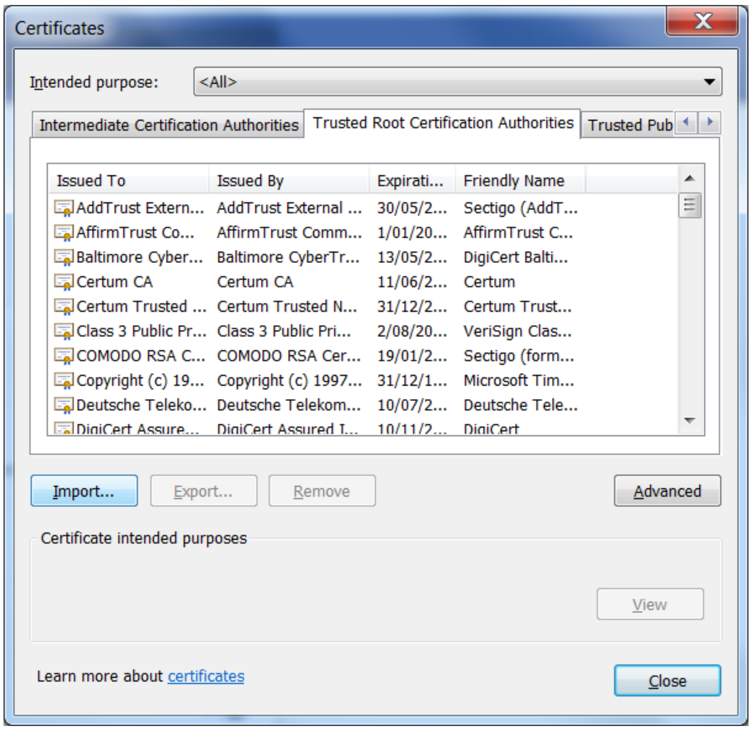
Preload pages for faster browsing and searching
Uses cookies to remember your preferences, even if you don't visit those pages

Manage certificates
Manage HTTPS/SSL certificates and settings

Site Settings
Control what information websites can use and what content they can show you

Clear browsing data
Clear history, cookies, cache, and more

Once the wizard reaches this screen, select the tab shown and click on the Import button:



Certificates

Intended purpose: <All>

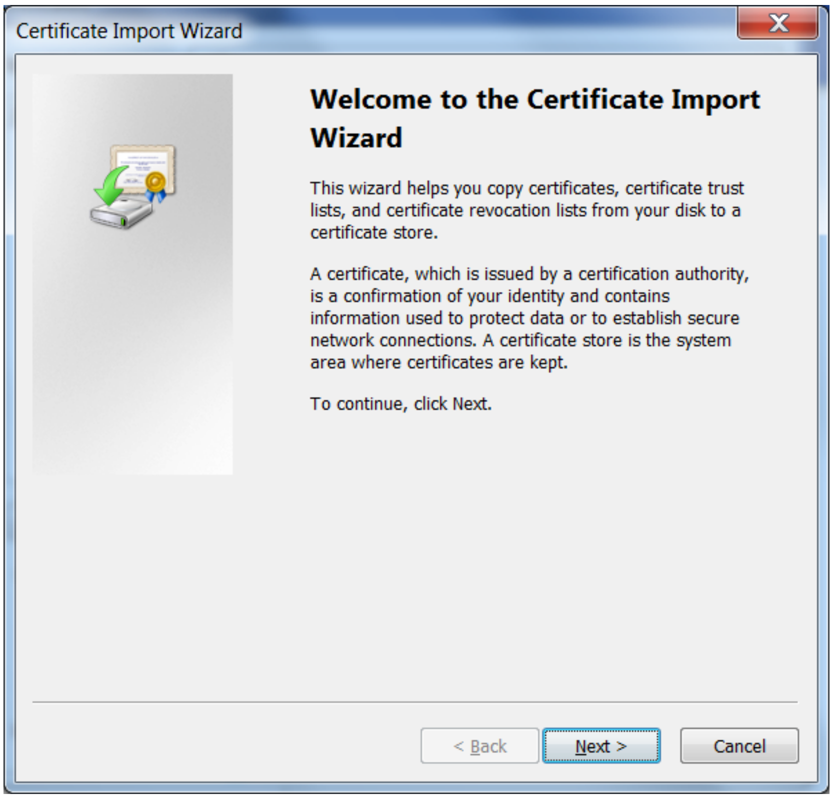
Intermediate Certification Authorities | Trusted Root Certification Authorities | Trusted Pub

Issued To	Issued By	Expirati...	Friendly Name
AddTrust Extern...	AddTrust External ...	30/05/2...	Sectigo (AddT...
AffirmTrust Co...	AffirmTrust Comm...	1/01/20...	AffirmTrust C...
Baltimore Cyber...	Baltimore CyberTr...	13/05/2...	DigiCert Balti...
Certum CA	Certum CA	11/06/2...	Certum
Certum Trusted ...	Certum Trusted N...	31/12/2...	Certum Trust...
Class 3 Public Pr...	Class 3 Public Pri...	2/08/20...	VeriSign Clas...
COMODO RSA C...	COMODO RSA Cer...	19/01/2...	Sectigo (form...
Copyright (c) 19...	Copyright (c) 1997...	31/12/1...	Microsoft Tim...
Deutsche Teleko...	Deutsche Telekom...	10/07/2...	Deutsche Tele...
DigiCert Assure...	DigiCert Assured I...	10/11/2...	DigiCert

Certificate intended purposes

Learn more about certificates

This will lead to the certificate import dialogue, which we see below:



Certificate Import Wizard

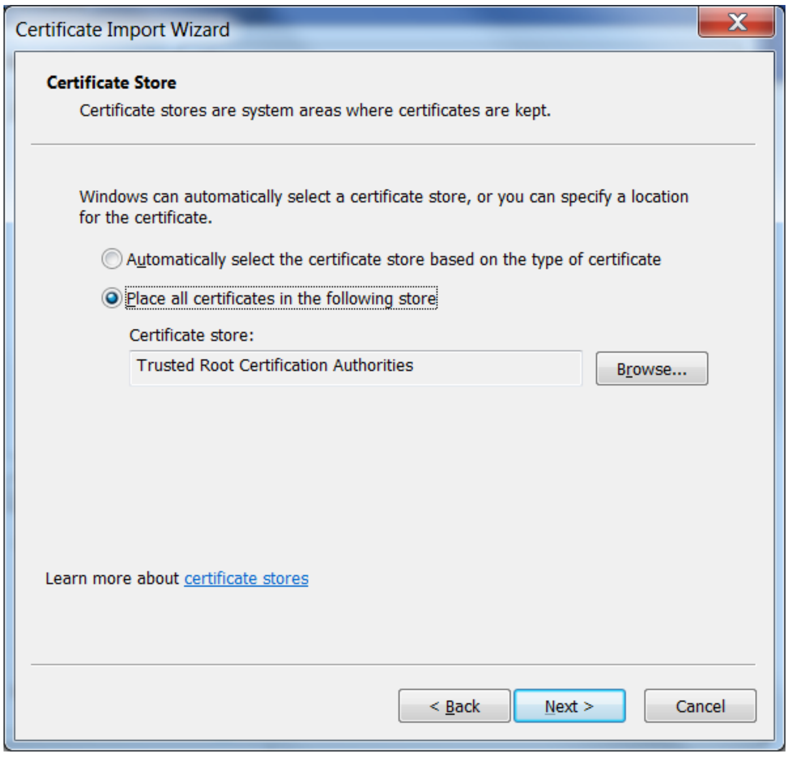
Welcome to the Certificate Import Wizard

This wizard helps you copy certificates, certificate trust lists, and certificate revocation lists from your disk to a certificate store.

A certificate, which is issued by a certification authority, is a confirmation of your identity and contains information used to protect data or to establish secure network connections. A certificate store is the system area where certificates are kept.

To continue, click Next.

Follow the workflow through the screens on the next page, accepting the defaults.


Certificate Import Wizard

Certificate Store

Certificate stores are system areas where certificates are kept.

Windows can automatically select a certificate store, or you can specify a location for the certificate.

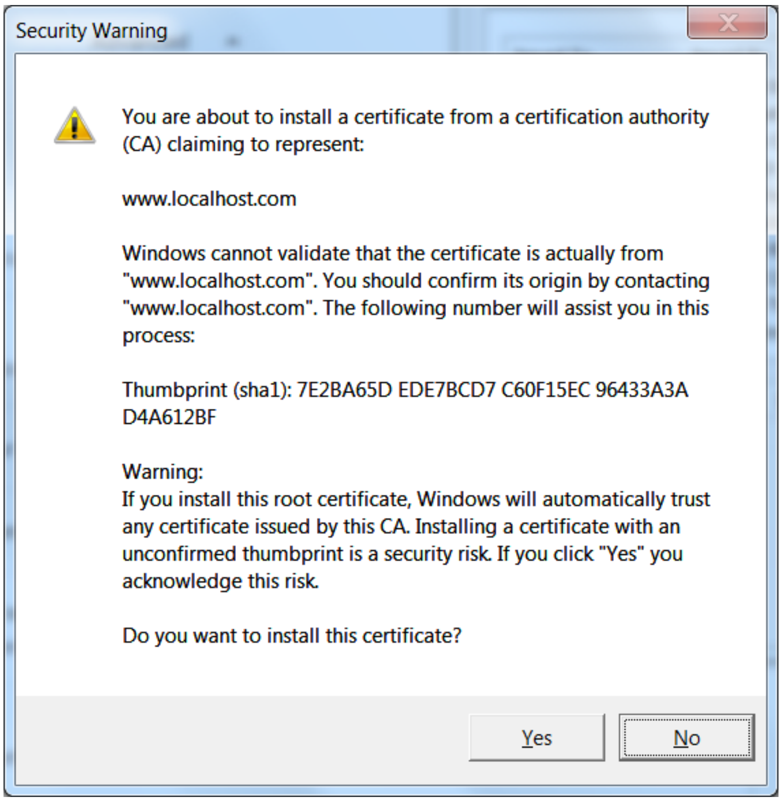
○ Automatically select the certificate store based on the type of certificate

● Place all certificates in the following store

Certificate store:

Trusted Root Certification Authorities

Learn more about certificate stores


Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

www.localhost.com

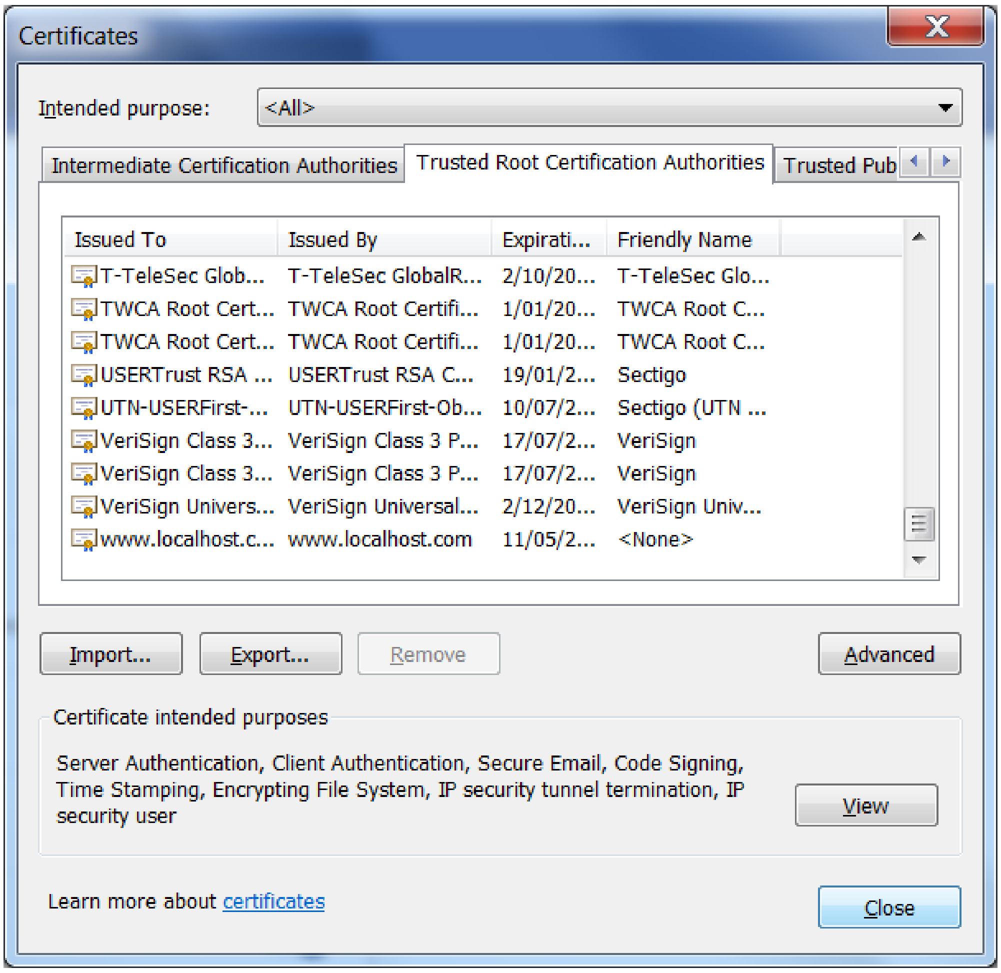
Windows cannot validate that the certificate is actually from "www.localhost.com". You should confirm its origin by contacting "www.localhost.com". The following number will assist you in this process:

Thumbprint (sha1): 7E2BA65D EDE7BCD7 C60F15EC 96433A3A D4A612BF

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

Evidently we are prepared to accept the risk, and effectively we are our own Certificate Authority. On the next page we see that localhost appears at the bottom of the screenshot amongst the Trusted Authorities.



Certificates

Intended purpose: <All>

Intermediate Certification Authorities | Trusted Root Certification Authorities | Trusted Pub

Issued To	Issued By	Expirati...	Friendly Name
T-TeleSec Glob...	T-TeleSec GlobalR...	2/10/20...	T-TeleSec Glo...
TWCA Root Cert...	TWCA Root Certifi...	1/01/20...	TWCA Root C...
TWCA Root Cert...	TWCA Root Certifi...	1/01/20...	TWCA Root C...
USERTrust RSA ...	USERTrust RSA C...	19/01/2...	Sectigo
UTN-USERFirst-...	UTN-USERFirst-Ob...	10/07/2...	Sectigo (UTN ...
VeriSign Class 3...	VeriSign Class 3 P...	17/07/2...	VeriSign
VeriSign Class 3...	VeriSign Class 3 P...	17/07/2...	VeriSign
VeriSign Univers...	VeriSign Universal...	2/12/20...	VeriSign Univ...
www.localhost.c...	www.localhost.com	11/05/2...	<None>

Import... Export... Remove Advanced

Certificate intended purposes

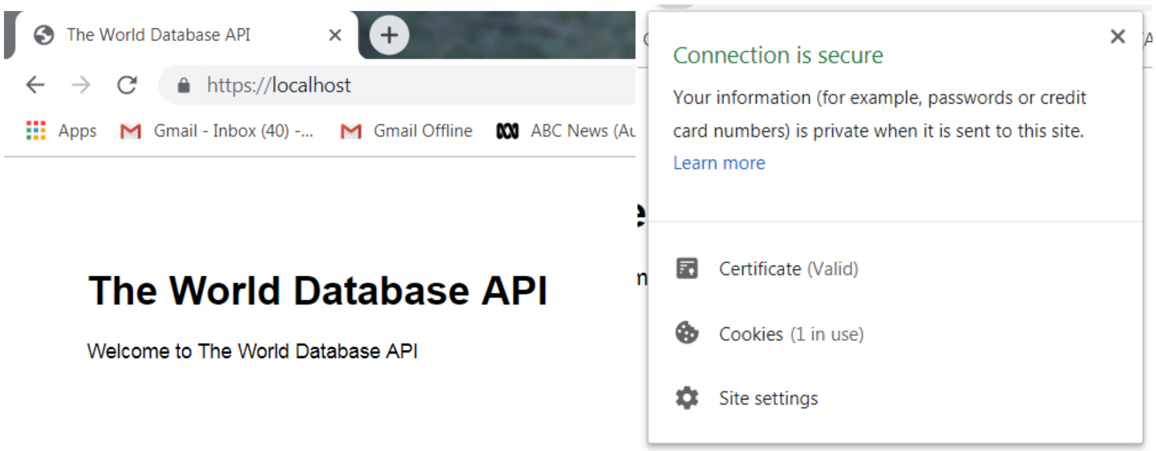
Server Authentication, Client Authentication, Secure Email, Code Signing, Time Stamping, Encrypting File System, IP security tunnel termination, IP security user

View

Learn more about certificates

Close

We now go ahead and restart the server, and we see that the site is now regarded as secure, with the padlock shown near the URL:



The World Database API

https://localhost

Apps Gmail - Inbox (40) -... Gmail Offline ABC News (Au

The World Database API

Welcome to The World Database API

Connection is secure

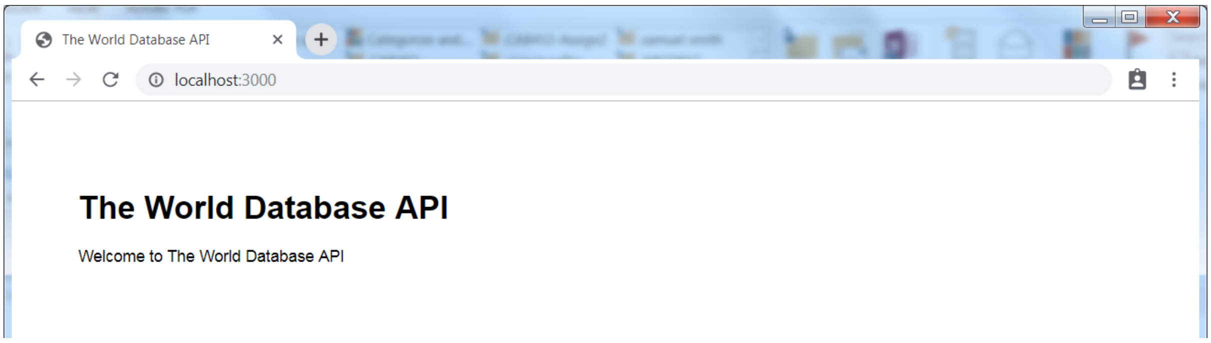
Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more

Certificate (Valid)

Cookies (1 in use)

Site settings

So far so good. This might not happen straight away – you may need to restart the browser or even reboot before this change takes effect, but it should work, and we have a trusted connection to our own server on localhost. You can confirm the other routes work ok, and serve their content via HTTPS. But what happens now if we go and load at http://localhost:3000? The answer is on the following page, and we see that everything works as before. [Note that this is based on an Express generated app. See below for discussion.]

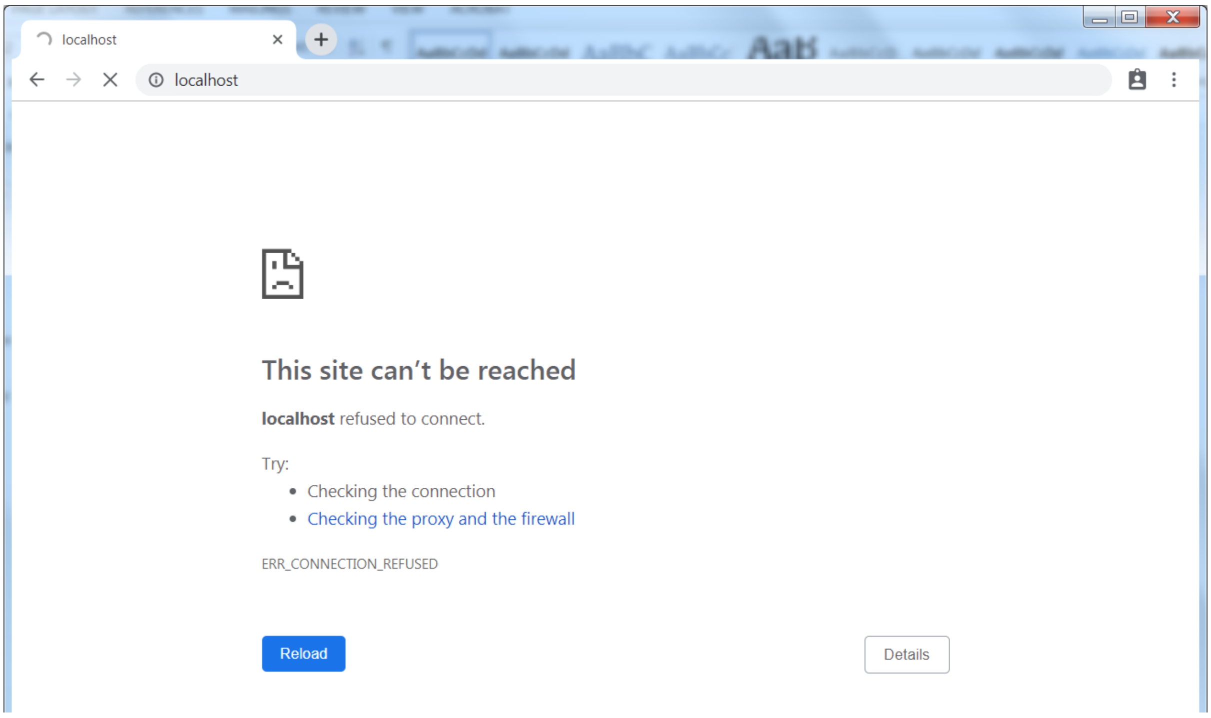

The World Database API

localhost:3000

The World Database API

Welcome to The World Database API

Essentially the page serves via HTTP as before. So perhaps we should understand our app a little better. Try it now with http://localhost. We see the following, showing that the site can't be reached via HTTP on the default port of 80.


localhost

localhost

This site can't be reached

localhost refused to connect.

Try:

- Checking the connection
- Checking the proxy and the firewall

ERR_CONNECTION_REFUSED

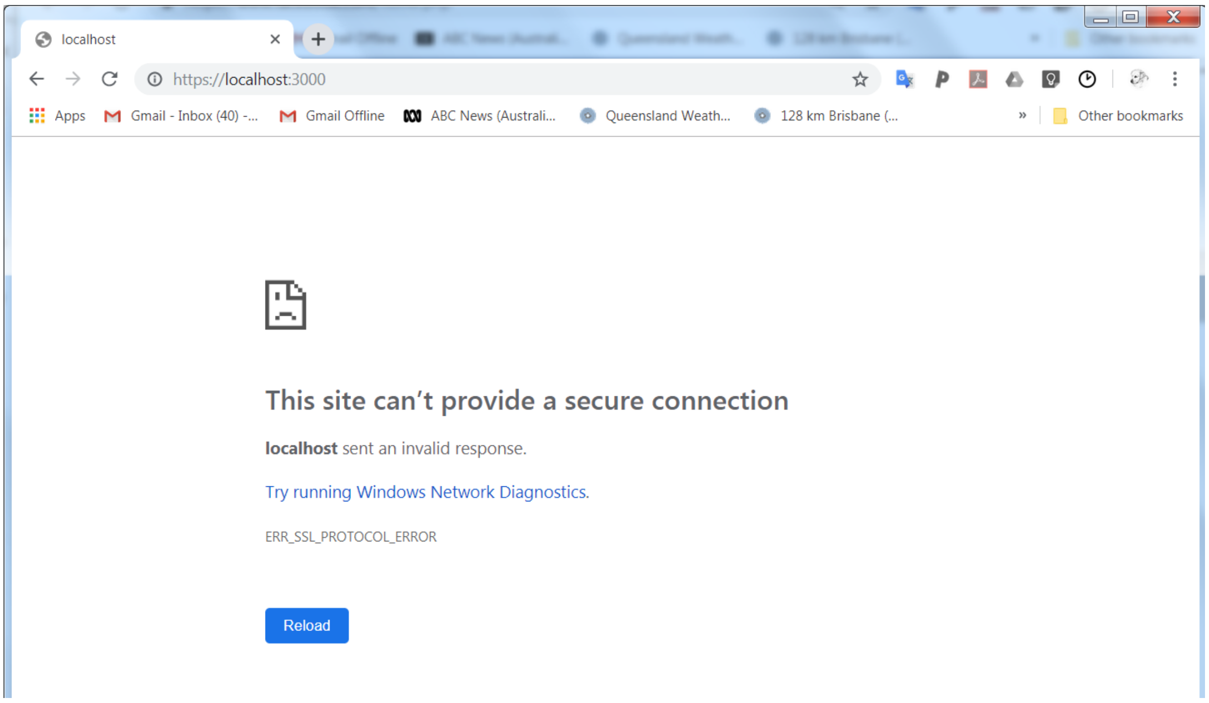
So we now take a closer look at the app itself. In the `bin` directory we will see a file called `www`. This is a node script which drives the creation of the server based on the app we have created in app.js. Open it up in your favourite editor and you will see the magic that has so far been creating the HTTP server that you have previously been using. That server is still in operation. So at the moment you are offering two servers – one on HTTP and one on HTTPS. This obviously undermines security. In practice, node deployments normally rely on a reverse proxy arrangement as discussed in the week 9 deployment lecture.

For now let's look at what happens when we use some standard security measures as provided by Helmet. The key aspect here is that Helmet ensures that the HSTS or Strict Transport Security header is present. This tells the client browser that the site should only be accessed via HTTPS rather than HTTP. You can find out more information at the MDN site: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security. Here we will let helmet take care of it for us. The process is very straightforward, requiring only two lines of code in `app.js`. We show a bit more code to show you where to place them:

```
const swaggerUI = require('swagger-ui-express');
const swaggerDocument = require('./docs/swagger.json');
const helmet = require('helmet');
```

```
app.use(logger('common'));

app.use(helmet());

// view engine setup
```

The effect of the HSTS header being set is to cause the server to try to serve HTTPS. Sometimes this works and sometimes it doesn't. Restart the server and give it a try on http://localhost:3000. We see that the server can't actually serve the page, as it can't serve HTTPS on port 3000:



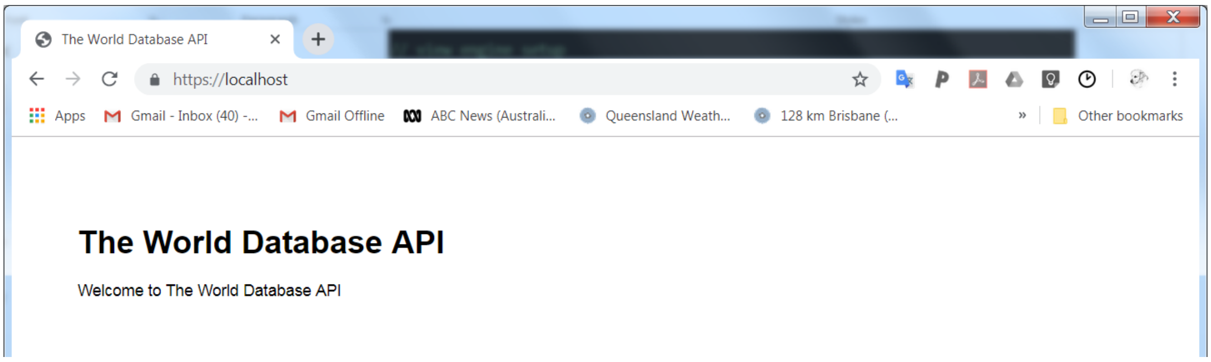
This site can't provide a secure connection

localhost sent an invalid response.

Try running Windows Network Diagnostics.

ERR_SSL_PROTOCOL_ERROR

But if we now try http://localhost, the result seems better.



The World Database API

Welcome to The World Database API

What we are seeing is that the http://localhost request on the default port is now redirected to be served via HTTPS. This isn't magic, though. Another port, say http://localhost:8000 will lead to a connection error.

There are a range of possible redirect strategies available if we want to maintain the previously advertised port at 3000 on HTTP. This is covered in a basic Express app context here:
https://stackoverflow.com/questions/7450940/automatic-https-connection-redirect-with-node-js-express

But the key problem in our case is that we are still running two servers. The least painful fix is to abandon port 3000. Just jump into the www file and edit this line of code to use the default HTTP port of 80 (it will be 3000 now):

```
var port = normalizePort(process.env.PORT || '80');
```

Then we will get a refused connection on http://localhost:3000 and, on Chrome at least, the HSTS header will cause the standard HTTP on port 80 to be redirected. This doesn't get rid of the problem, but prevents an unsecured connection on port 3000.

The lesson here is that Express generator is brilliant as a starting point, but over time we will learn to have our own application templates and to hack from those. Express is very powerful and as we start to understand better how it works, we will learn that we don't have to follow everything in the generated model.

If we edit `www`, then the app is left alone for specifying the operations and middleware usage. This is perhaps the cleanest of the options open to us. After migrating the code from `app.js`, our file looks like the listing below. Note that use of the double dot for the `require('../app')` and the single dot for `'./sslcert/cert.key'` in `fs.readFileSync()` is not an error. The path in the `fs` usage is relative to the current working directory of the process – here the application root. The `require` function has the context of the current file. So this is *very* confusing, but correct.

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var app = require('../app');
var debug = require('debug')('expworld:server');
const https = require('https');
const fs = require('fs');
const privateKey = fs.readFileSync('./sslcert/cert.key','utf8');
const certificate = fs.readFileSync('./sslcert/cert.pem','utf8');
const credentials = {
  key: privateKey,
  cert: certificate
};

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '443');
app.set('port', port);

/**
 * Create HTTPS server.
 */

const server = https.createServer(credentials,app);
```

The remainder of the file is unchanged. Save and restart the server and we will see that HTTP connections are refused if the port is non-standard, the bare http://localhost is redirected to https://localhost, and HTTPS works as before. If we try another browser, we may find that the behaviour is different, and that all HTTP requests are refused. This difference is due to Chrome preloading the HSTS headers. If we want to be sure that that the HTTP is properly redirected, we need to use an approach like that linked from Stack Overflow above, but we will leave that as an exercise for those interested.