What Does a Python HTTPS Application Look Like?

Now that you have an understanding of the basic parts required for a making a Python HTTPS application, it's time to tie all the pieces together one-by-one to your application from before. This will ensure that your communication between server and client is secure.

It's possible to set up the entire PKI infrastructure on your own machine, and this is exactly what you'll be doing in this section. It's not as hard as it sounds, so don't worry! Becoming a real Certificate Authority is significantly harder than taking the steps below, but what you'll read is, more or less, all you'd need to run your own CA.

Becoming a Certificate Authority

A Certificate Authority is nothing more than a very important public and private key pair. To become a CA, you just need to generate a public and private key pair.

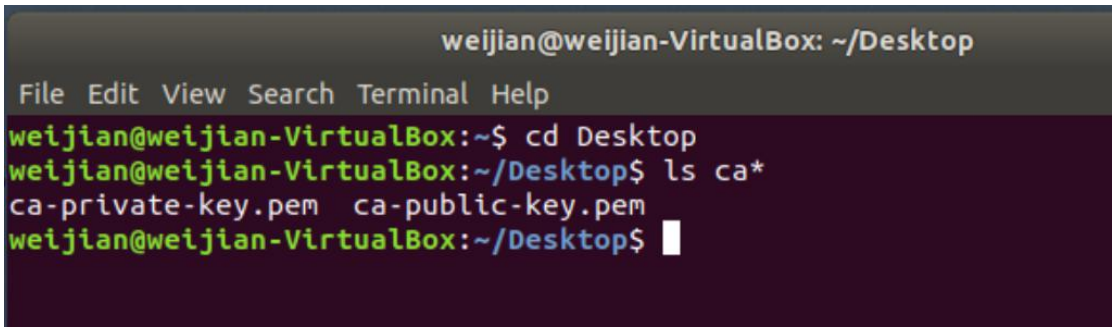
```
# pki_helpers.py
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from datetime import datetime, timedelta
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes


def generate_private_key(filename: str, passphrase: str):
    private_key = rsa.generate_private_key(
        public_exponent=65537, key_size=2048, backend=default_backend()
    )

    utf8_pass = passphrase.encode("utf-8")
    algorithm = serialization.BestAvailableEncryption(utf8_pass)

    with open(filename, "wb") as keyfile:
        keyfile.write(
            private_key.private_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PrivateFormat.TraditionalOpenSSL,
                encryption_algorithm=algorithm,
            )
        )

    return private_key
```

```
def generate_public_key(private_key, filename, **kwargs):
    subject = x509.Name(
        [
            x509.NameAttribute(NameOID.COUNTRY_NAME, kwargs["country"]),
            x509.NameAttribute(
                NameOID.STATE_OR_PROVINCE_NAME, kwargs["state"]
            ),
            x509.NameAttribute(NameOID.LOCALITY_NAME, kwargs["locality"]),
            x509.NameAttribute(NameOID.ORGANIZATION_NAME,
kwargs["org"]),
            x509.NameAttribute(NameOID.COMMON_NAME,
kwargs["hostname"]),
        ]
    )

    # Because this is self signed, the issuer is always the subject
    issuer = subject

    # This certificate is valid from now until 30 days
    valid_from = datetime.utcnow()
    valid_to = valid_from + timedelta(days=30)

    # Used to build the certificate
    builder = (
        x509.CertificateBuilder()
        .subject_name(subject)
        .issuer_name(issuer)
        .public_key(private_key.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(valid_from)
        .not_valid_after(valid_to)
    )

    # Sign the certificate with the private key
    public_key = builder.sign(
        private_key, hashes.SHA256(), default_backend()
    )

    with open(filename, "wb") as certfile:
        certfile.write(public_key.public_bytes(serialization.Encoding.PEM))

    return public_key
```

We compile this code to generate your private and public key pair:

```
from pki_helpers import generate_private_key, generate_public_key
>>> private_key = generate_private_key("ca-private-key.pem", "secret_password")
>>> private_key
>>> generate_public_key(
...     private_key,
...     filename="ca-public-key.pem",
...     country="US",
...     state="Maryland",
...     locality="Baltimore",
...     org="My CA Company",
...     hostname="my-ca.com",
... )
```

After importing your helper functions from pki_helpers, you first generate your private key and save it to the file ca-private-key.pem. You then pass that private key into generate_public_key() to generate your public key. In your directory you should now have two files:



Trusting Your Server

The first step to your server becoming trusted is for you to generate a Certificate Signing Request (CSR). In the real world, the CSR would be sent to an actual Certificate Authority like Verisign or Let's Encrypt. In this example, you'll use the CA you just created.

Paste the code for generating a CSR into the pki_helpers.py file from above:

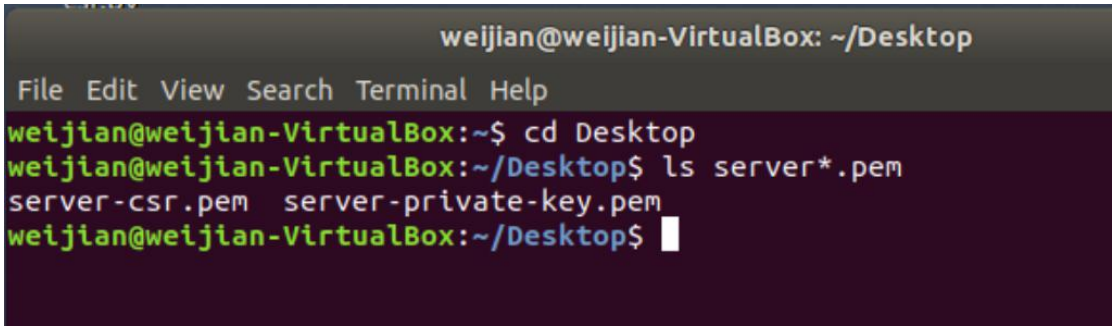
```
# pki_helpers.py
def generate_csr(private_key, filename, **kwargs):
    subject = x509.Name(
        [
            x509.NameAttribute(NameOID.COUNTRY_NAME, kwargs["country"]),
            x509.NameAttribute(
                NameOID.STATE_OR_PROVINCE_NAME, kwargs["state"]
            ),
            x509.NameAttribute(NameOID.LOCALITY_NAME, kwargs["locality"]),
            x509.NameAttribute(NameOID.ORGANIZATION_NAME, kwargs["org"]),
            x509.NameAttribute(NameOID.COMMON_NAME, kwargs["hostname"]),
```

```
        ]
    )

    # Generate any alternative dns names
    alt_names = []
    for name in kwargs.get("alt_names", []):
        alt_names.append(x509.DNSName(name))
    san = x509.SubjectAlternativeName(alt_names)

    builder = (
        x509.CertificateSigningRequestBuilder()
        .subject_name(subject)
        .add_extension(san, critical=False)
    )

    csr = builder.sign(private_key, hashes.SHA256(), default_backend())

    with open(filename, "wb") as csrfile:
        csrfile.write(csr.public_bytes(serialization.Encoding.PEM))

    return csr
```

You'll notice that, in order to create a CSR, you'll need a private key first. Luckily, you can use the same generate_private_key() from when you created your CA's private key. Using the above function and the previous methods defined, you can do the following:

```
from pki_helpers import generate_csr, generate_private_key
>>> server_private_key = generate_private_key(
...     "server-private-key.pem", "serverpassword"
... )
>>> server_private_key
>>> generate_csr(
...     server_private_key,
...     filename="server-csr.pem",
...     country="US",
...     state="Maryland",
...     locality="Baltimore",
...     org="My Company",
...     alt_names=["localhost"],
...     hostname="my-site.com",
... )
```

You can view your new CSR and private key from the console:



Since you are the CA in this case, you can forego that headache create your very own verified public key. To do that, you'll add another function to your pki_helpers.py file:

```
# pki_helpers.py
def sign_csr(csr, ca_public_key, ca_private_key, new_filename):
    valid_from = datetime.utcnow()
    valid_until = valid_from + timedelta(days=30)

    builder = (
        x509.CertificateBuilder()
        .subject_name(csr.subject)
        .issuer_name(ca_public_key.subject)
        .public_key(csr.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(valid_from)
        .not_valid_after(valid_until)
    )

    for extension in csr.extensions:
        builder = builder.add_extension(extension.value, extension.critical)

    public_key = builder.sign(
        private_key=ca_private_key,
        algorithm=hashes.SHA256(),
        backend=default_backend(),
    )

    with open(new_filename, "wb") as keyfile:
        keyfile.write(public_key.public_bytes(serialization.Encoding.PEM))
```

The next step is to fire up the Python console and use sign_csr(). You'll need to load your CSR and your CA's private and public key. Begin by loading your CSR:

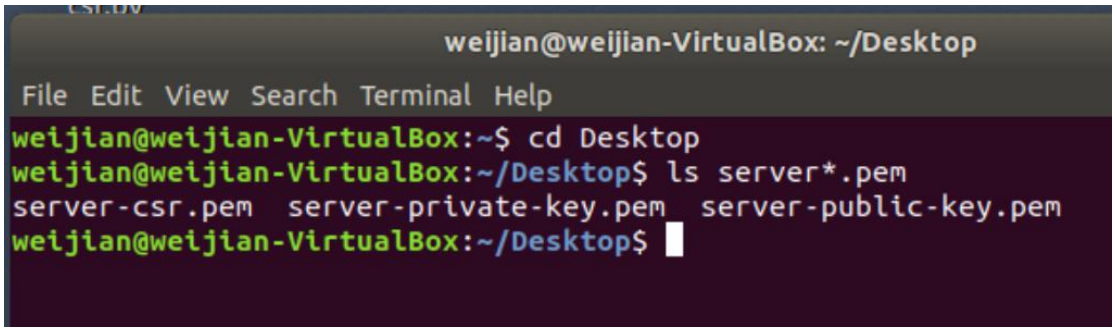
```
>>> from cryptography import x509
```

```
>>> from cryptography.hazmat.backends import default_backend
>>> csr_file = open("server-csr.pem", "rb")
>>> csr = x509.load_pem_x509_csr(csr_file.read(), default_backend())
>>> csr

>>> ca_public_key_file = open("ca-public-key.pem", "rb")
>>> ca_public_key = x509.load_pem_x509_certificate(
...     ca_public_key_file.read(), default_backend()
... )
>>> ca_public_key

>>> from getpass import getpass
>>> from cryptography.hazmat.primitives import serialization
>>> ca_private_key_file = open("ca-private-key.pem", "rb")
>>> ca_private_key = serialization.load_pem_private_key(
...     ca_private_key_file.read(),
...     getpass().encode("utf-8"),
...     default_backend(),
... )
Password:
>>> private_key
```

->Here, password is “sercret_password”.

```
>>> from pki_helpers import sign_csr
>>> sign_csr(csr, ca_public_key, ca_private_key, "server-public-key.pem")
```

After running this, you should have three server key files in your directory:



Whew! That was quite a lot of work. The good news is that now that you have your private and public key pair, you don’t have to change any server code to start using it.

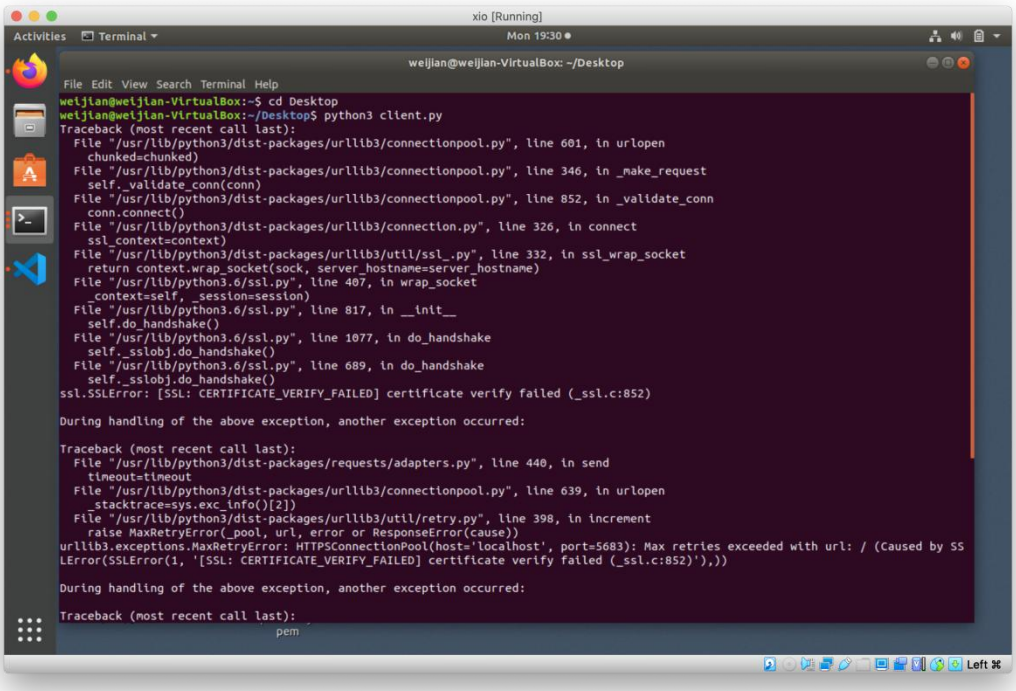
Using your original server.py file, run the following command to start your brand new Python HTTPS application:

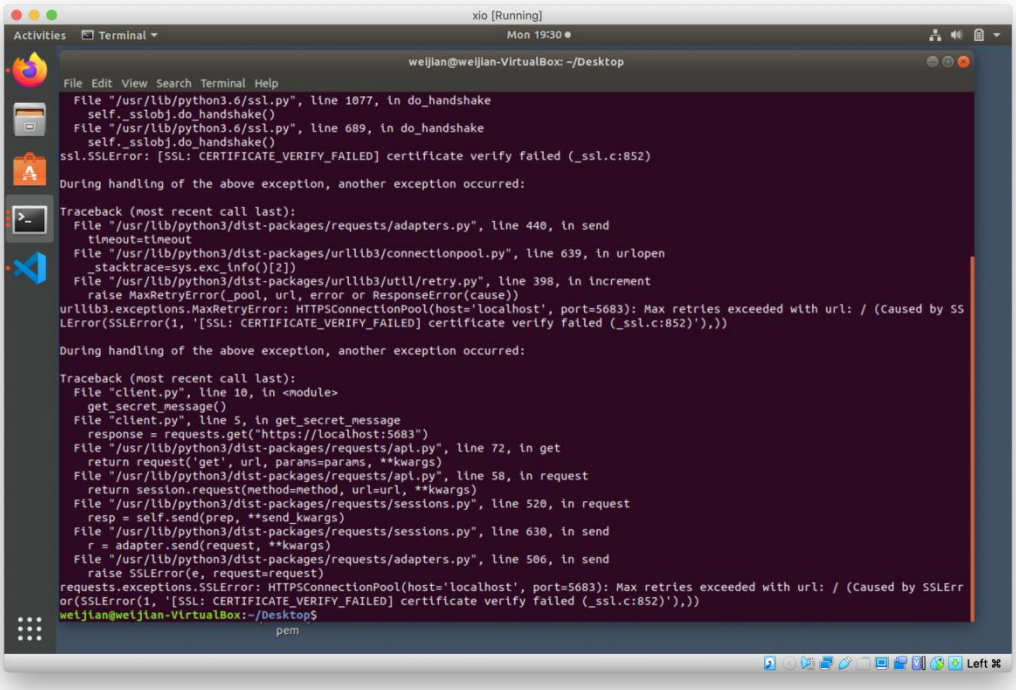
```
uwsgi --http-socket localhost:5683 --mount /=server:app
```

Congratulations! You now have a Python HTTPS-enabled server running with your very own private-public key pair, which was signed by your very own Certificate Authority! Now, all that's left to do is query your server. First, you'll need to make some changes to the client.py code:

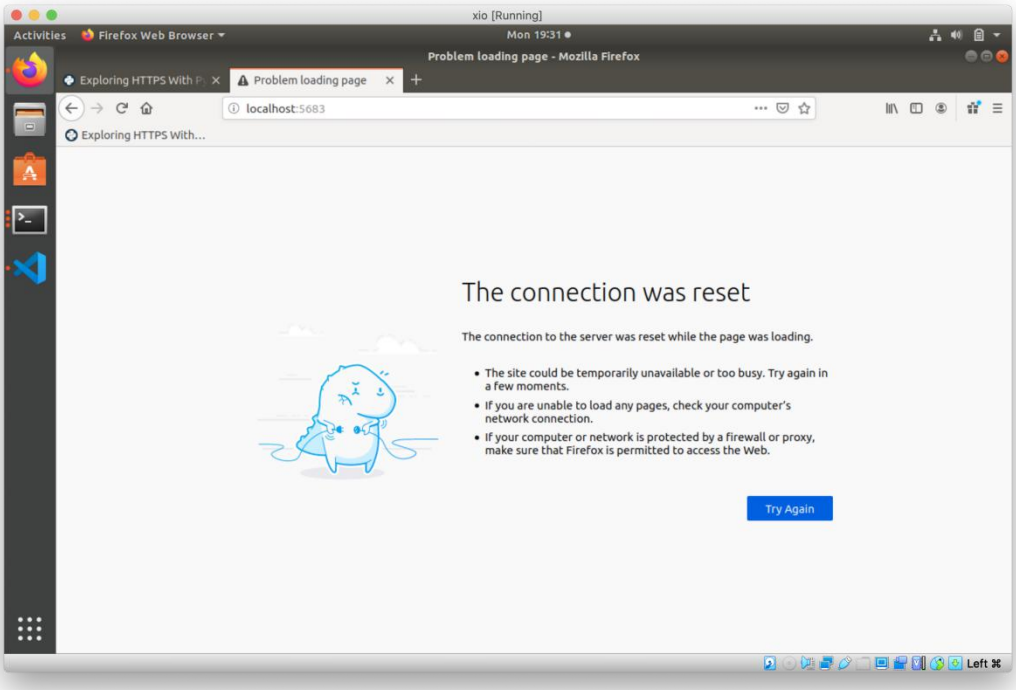
```
# client.py
import os
import requests

def get_secret_message():
    response = requests.get("https://localhost:5683")
    print(f"The secret message is {response.text}")

if __name__ == "__main__":
    get_secret_message()
```

The only change from the previous code is from http to https. If you try to run this code, then you'll be met with an error:





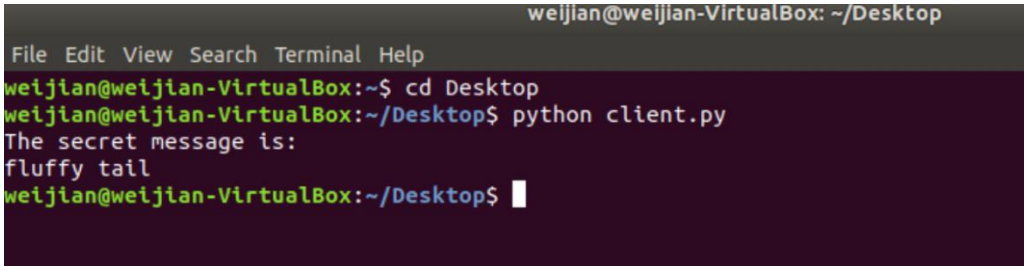
If you attempt to navigate to your website with your browser, then you'll get a similar message:



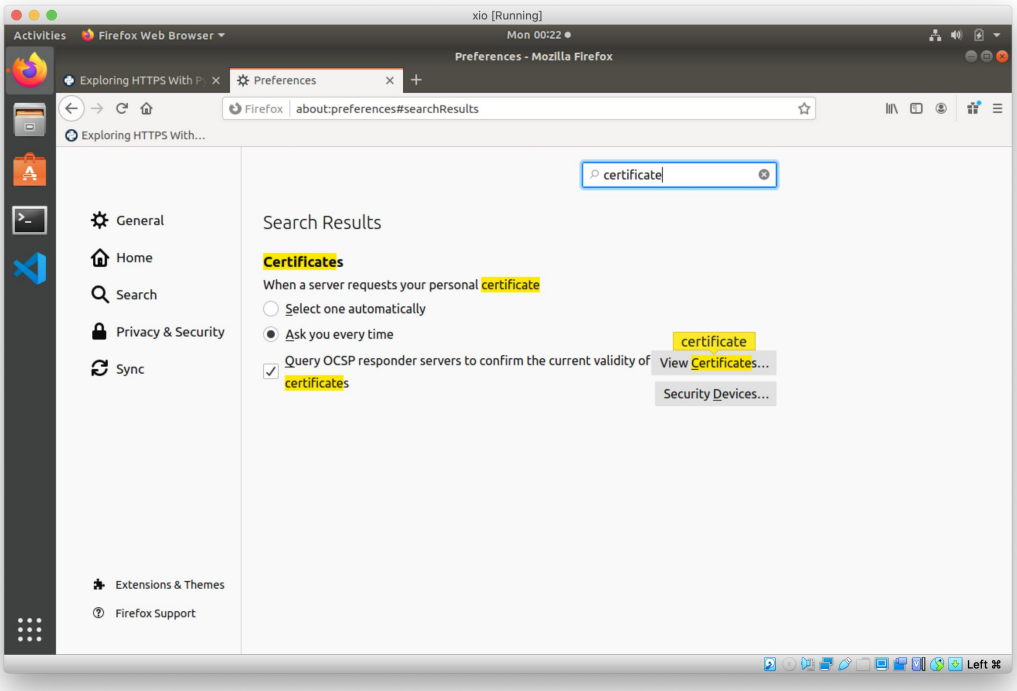
If you want to avoid this message, then you have to tell requests about your Certificate Authority! All you need to do is point requests at the ca-public-key.pem file that you generated earlier:

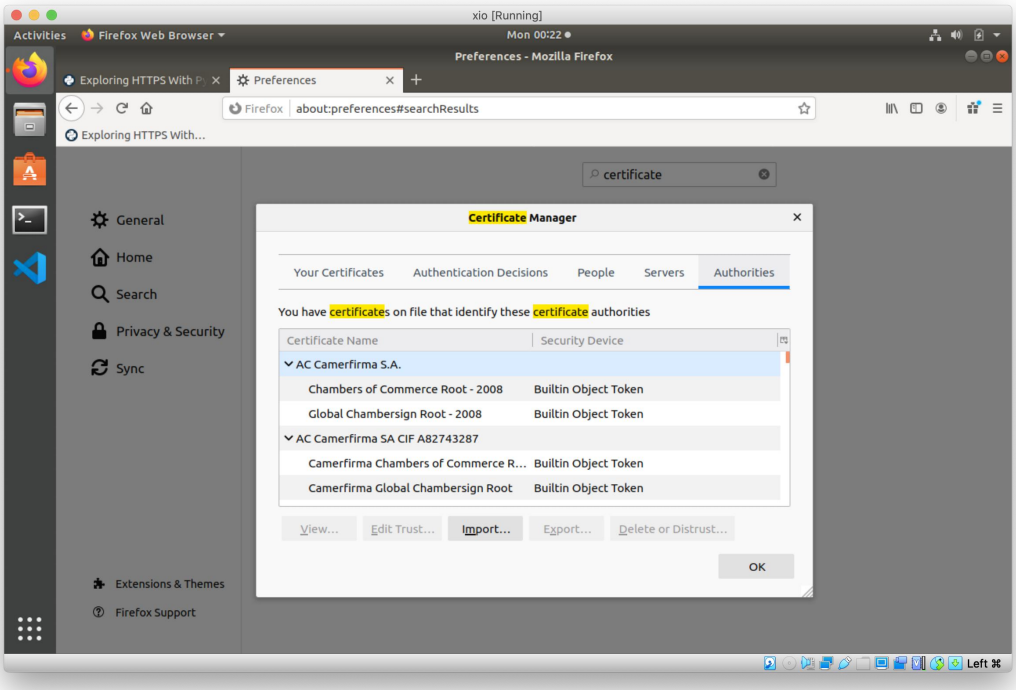
```
# client.py
def get_secret_message():
    response = requests.get("http://localhost:5683", verify="ca-public-key.pem")
    print(f"The secret message is {response.text}")
```

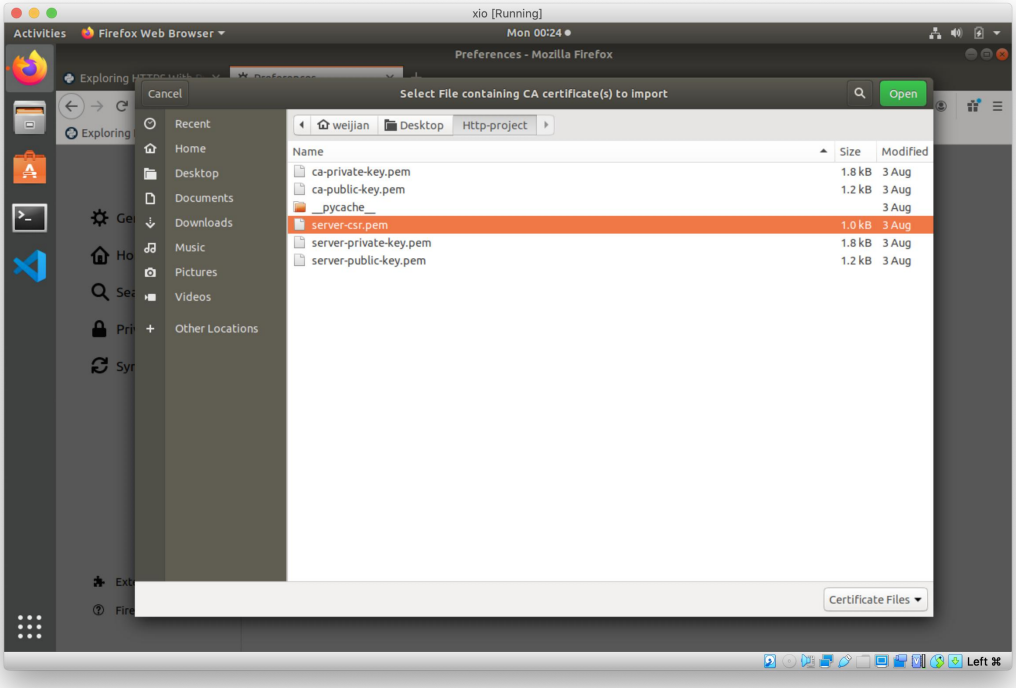
After doing that, you should be able to run the following successfully:



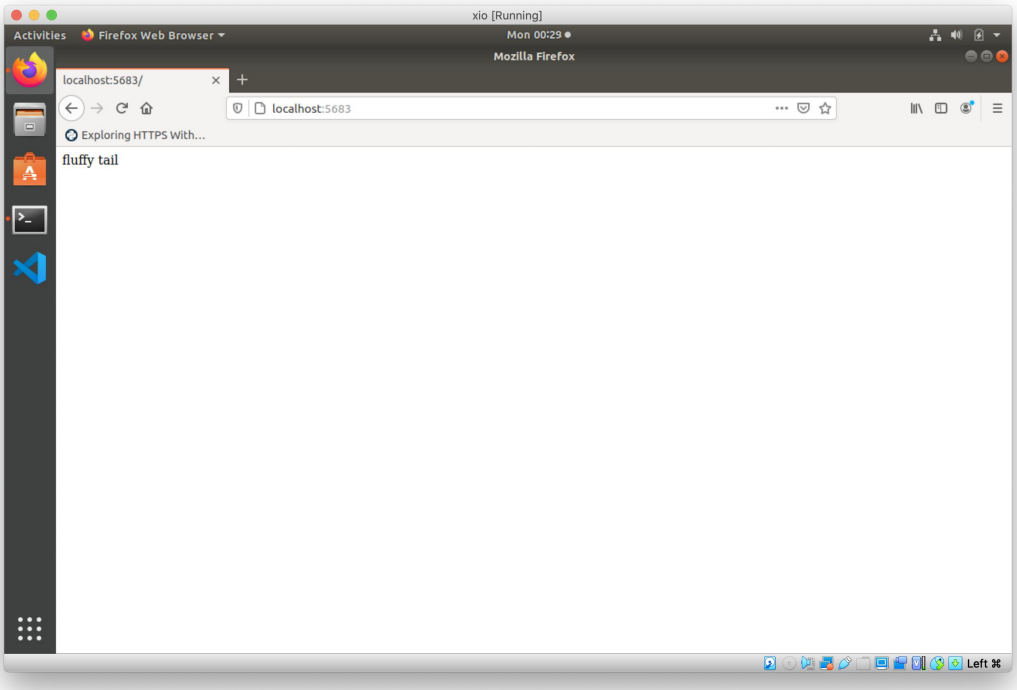
After we created certificate, now we import it to our browse.







After we import certificate, run the server, then go to localhost:5683. Now we can see the secret message on the browser.


xio [Running]
Activities
Firefox Web Browser
Mon 00:29
Mozilla Firefox
localhost:5683/
localhost:5683
Exploring HTTPS With...
fluffy tail
Left ⌘