# Setup SSL/HTTPS on JBoss Wildfly Application Server

In this tutorial, we will configure SSL/HTTPS on the JBoss Wildfly application server.

For the purposes of this tutorial, we will be working with a preinstalled **Wildfly 16.0.0.Final (Java EE Full & Web Distribution).**

To learn how to download and setup Wildfly, check out my article on this.

**Install JBoss Wildfly on Ubuntu 18.04**
WildFly, formerly known as JBoss AS, or simply JBoss, is a Java EE certified application server authored by JBoss, now…
medium.com

Before we continue, let us briefly discuss how SSL certificates work.

A **Certificate Authority (CA)** like DigiCert issues you a digitally signed certificate against your **Certificate Signing Request (CSR)**. The CA scrutinizes the details provided in the CSR and once it deems everything is fine, it signs your SSL certificate with the private key from its root and sends it back along with maybe one or more **Intermediate CA Certificates**.

These Intermediate CA Certificates combined with your digitally signed end-user SSL certificate form what is called a Certificate Chain.

A **Certificate Chain** is an ordered list of certificates, containing an end-user SSL Certificate and Intermediate CA Certificates, that enables the receiver to verify that the sender and all CA's are trustworthy. The chain or path begins with the end-user SSL certificate, and each certificate in the chain is signed by the entity identified by the next certificate in the chain.

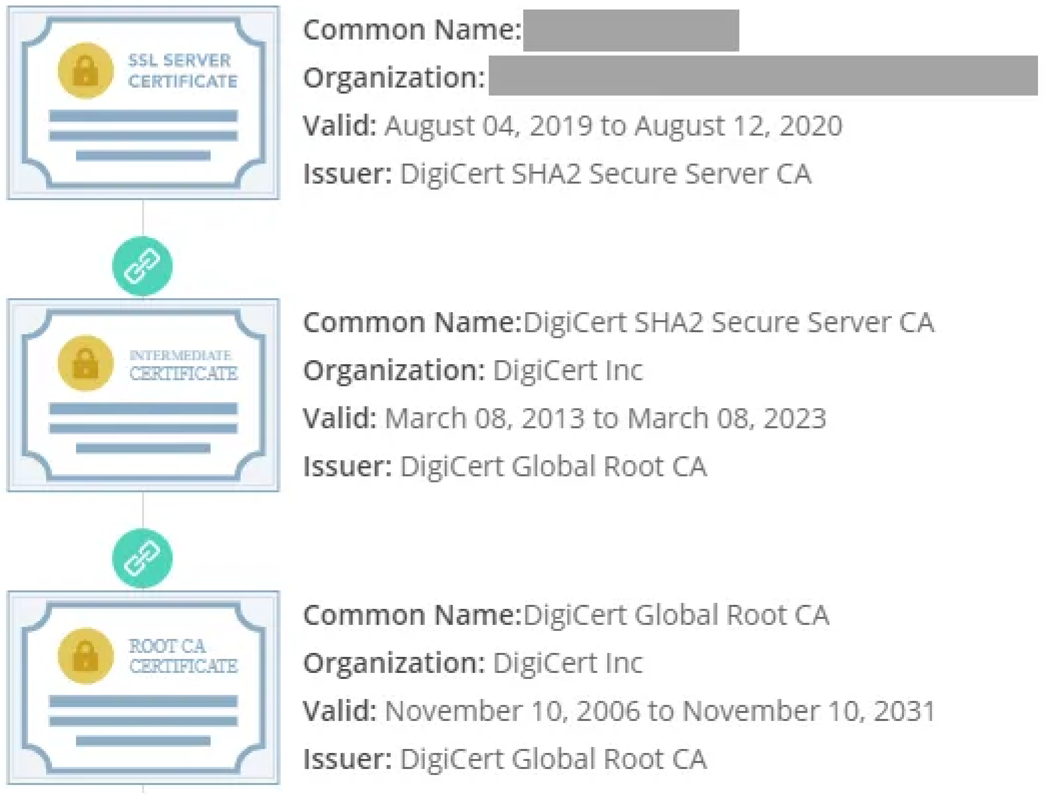
For a client like your browser, when it receives a Certificate Chain (end-user SSL certificate along with any intermediate CA certificates) from the server as part of SSL handshake, the SSL certificate validation is done by starting from the bottom of the Certificate Chain (end-user SSL certificate) and going all the way up to the root intermediate CA certificate. If the issuer of the root intermediate CA certificate is one of the trusted CA roots in the client's trust store, the client derives that the Certificate Chain offered to it by the server is trustful.

This is intuitive because each intermediate CA certificate is digitally signed by a trusted issuer which makes it trustful. This chain of trust starts from the topmost/root intermediate CA certificate that is digitally signed by a globally trusted Root CA's private key and then goes all the way down to the end-user SSL certificate, with each subsequent intermediate CA certificate getting signed by its upstream trusted issuer's private key. Thus in this way, the whole Certificate Chain becomes trustful.

If however along the validation path, the issuer of the topmost root intermediate certificate is not present in the client's trust store then the Certificate Chain is considered broken and rejected and the SSL communication is aborted by the client with an error or warning indicating to the user that the communication is not secured with SSL/HTTPS because the certificate chain is not valid.

Now consider the SSL certificate that I received from a DigiCert subsidiary against my CSR. Let us call this Digicert subsidiary **ABC.** Along with the end-user SSL certificate, I also got back an intermediate CA certificate from DigiCert. I set up the certificate on my server and then used this very nice online SSL utility to visualize the Certificate Chain that my end-user SSL certificate carries.

Below is a screenshot of what I got



**Common Name:** [redacted]
**Organization:** [redacted]
**Valid:** August 04, 2019 to August 12, 2020
**Issuer:** DigiCert SHA2 Secure Server CA

**Common Name:**DigiCert SHA2 Secure Server CA
**Organization:** DigiCert Inc
**Valid:** March 08, 2013 to March 08, 2023
**Issuer:** DigiCert Global Root CA

**Common Name:**DigiCert Global Root CA
**Organization:** DigiCert Inc
**Valid:** November 10, 2006 to November 10, 2031
**Issuer:** DigiCert Global Root CA

As we can see, my end-user SSL certificate was digitally signed using an intermediate CA certificate that was signed and issued by **DigiCert SHA2 Secure Server CA** to my DigiCert subsidiary **ABC**. The **DigiCert SHA2 Secure Server CA** is an Intermediary Root CA of DigiCert.

The intermediate certificate used by **DigiCert SHA2 Secure Server CA,** for signing the intermediate certificate issued to **ABC,** was signed and issued by **DigiCert Global Root CA** which is the Root CA of DigiCert.

As DigiCert Root CA is a widely trusted CA and because the signatures of all intermediate CA certificates in the chain should verify up to the Root CA Certificate, this forms a chain of trust that can be trusted by any client application willing to communicate with my server over SSL/HTTPS.

So that was a brief overview of how SSL certificates work. Let us continue with the tutorial.

## Setup SSL/HTTPS on JBoss Wildfly

The digitally signed certificate(s) returned by the CA can be in any accepted format but the **PEM** format is the most common format that CA issue certificates in. The **PEM** certificates usually have extensions such as **.pem, .crt, .cer, and .key**. They are Base64 encoded ASCII files and contain " **-----BEGIN CERTIFICATE -----**" and "**-----END CERTIFICATE -----**" statements. SSL certificates, root anf intermediate CA certificates, and private keys can all be put into the PEM format.

The CA issued certificates we will be using for this tutorial are in PEM format with **.crt** extension.

Now to setup SSL on Wildfly, we first need to create a Keystore file. A **Keystore** file is used to store cryptographic keys and certificates. There are three kinds of entries that can be stored in a Keystore file depending upon the type of Keystore file it is.

**Private Key:** This is a type of key that is used in asymmetric cryptography. It is usually protected with a password because of its sensitivity. It can also be used to sign a digital signature.

**Certificate:** A certificate contains a public key that can identify the subject claimed in the certificate. It is usually used to verify the identity of a server.

**Secret Key:** A key entry that is used in symmetric cryptography.

Now since Wildfly is written in Java, we will select a Keystore file type that is compatible with Java.

A few different types of Keystore files for Java are **JKS, JCEKS, PKCS#12, PKCS#11,** and **DKS**. For this tutorial, we will go with the **PKCS#12** Keystore file.

The **PKCS#12** is a standard portable Keystore file type that can be used with Java and other languages. The **PKCS#12** or **PFX** format is a binary format for storing the server certificate, any Root or Intermediate CA certificates, and the private key in one encryptable file. These Keystore files usually have extensions such as **.pfx and .p12**.

Let us create a **PKCS#12** Keystore file that will hold our server's private key (generated when we created the Certificate Signing Request ), the end-user SSL certificate, and the intermediate CA certificate.

Copy the private key and the certificate files to the Wildfly configuration directory (***${WILDFLY_HOME}/standalone/configuration)*** and run the below command while in the same directory.

```
$ sudo openssl pkcs12 -export -out wildfly-pkcs12.pfx -in mydomain.crt -inkey server-prv.key -certfile ca-cert.crt
```

When we run the above command. we will be prompted to enter a password to secure our **PKCS#12** Keystore file. Enter the desired password and continue. A "***.pfx***" file by the name `wildfly-pkcs12.pfx` will be created in the current directory.

We have created the **PKCS#12** Keystore file using the **OpenSSL** library.

---

*The OpenSSL is a general-purpose cryptography library that provides an open-source implementation of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols.*

*It includes utility functions for generating* ***RSA private keys****,* ***Certificate Signing Requests (CSRs)****,* ***checksums****,* ***managing certificates*** *and* ***performing encryption/decryption****. It is also used to convert certificates in one format into another and also to create various types of Keystore files.*

---

Before we continue further, let us discuss the different attributes of the **OpenSSL** command that we used above to generate a **PKCS#12** Keystore file.

- `-pkcs12` : Use the file utility for **PKCS#12** files in OpenSSL.
- `-export -out wildfly-pkcs12.pfx` : Export and save the **PKCS#12** file as **wildfly-pkcs12.pfx**. The file extension can also be ***".p12"***.
- `-in mydomain.crt` : The end-user SSL certificate returned by the certificate signing authority.

- `-inKey server-prv.key` : The private key of the server that was generated when the CSR was created.
- `-certfile ca-cert.crt` : This is optional, this is if we have any additional certificates we would like to include in the **PKCS#12** Keystore file. In our case, we do have an intermediate CA certificate that was returned by the certificate signing authority along with our end-user SSL certificate.

Once the Keystore file is created, we now need to configure the Wildfly server for SSL/HTTPS.

While in the Wildfly configuration directory, open the `standalone.xml` file and make the following changes.

Add a new ***<security-realm>*** element by the name "***UndertowRealm"*** under the **<security-realms>** element**.** Specify the Keystore file path along with the name, parent directory path, and password as shown below.

```
<security-realm name="UndertowRealm">
  <server-identities>
     <ssl>
       <keystore path="wildfly-pkcs12.pfx" relative-to="jboss.server.config.dir" keystore-password="<your-
password>" />
     </ssl>
   </server-identities>
 </security-realm>
```

The attribute `relative-to` is where we specify the parent directory path to which the value of `path` attribute is relative to. Its value is `jboss.server.config.dir` which is a property placeholder for the Wildfly configuration directory and translates to ${***WILDFLY_HOME}/wildfly/standalone/configuration/***.

Now locate the configuration for the ***undertow*** subsystem in your `standalone.xml` and make changes to the ***<https-listener>*** element as shown below.

```
<subsystem xmlns="urn:jboss:domain:undertow:8.0" default-server="default-server" default-virtual-host="default-
host" default-servlet-container="default" default-security-domain="other" statistics-
enabled="${wildfly.undertow.statistics-enabled:${wildfly.statistics-enabled:false}}">

  <buffer-cache name="default"/>
  <server name="default-server">
    <http-listener name="default" socket-binding="http" redirect-socket="https" enable-http2="true"/>
    <https-listener name="https" socket-binding="https" security-realm="UndertowRealm" enable-http2="true"/>
    <host name="default-host" alias="localhost">
      <location name="/" handler="welcome-content"/>
      <http-invoker security-realm="ApplicationRealm"/>
    </host>
  </server>
  <servlet-container name="default">
    <jsp-config/>
    <websockets/>
  </servlet-container>
  <handlers>
    <file name="welcome-content" path="${jboss.home.dir}/welcome-   content"/>
  </handlers>
</subsystem>
```

We have set the value of `security-realm` attribute of ***<https-listener>*** element to **UndertowRealm,** that we created before.

Save the `standalone.xml` and restart the Wildfly server.

Wildfly by default listens for HTTPS traffic on TCP port **8443**. However, the standard TCP port used by HTTPS connections is **443.** We can use the **iptables** utility to configure the Linux kernel firewall and redirect HTTPS traffic on port **443** to port **8443** that is used by Wildfly for HTTPS communication.

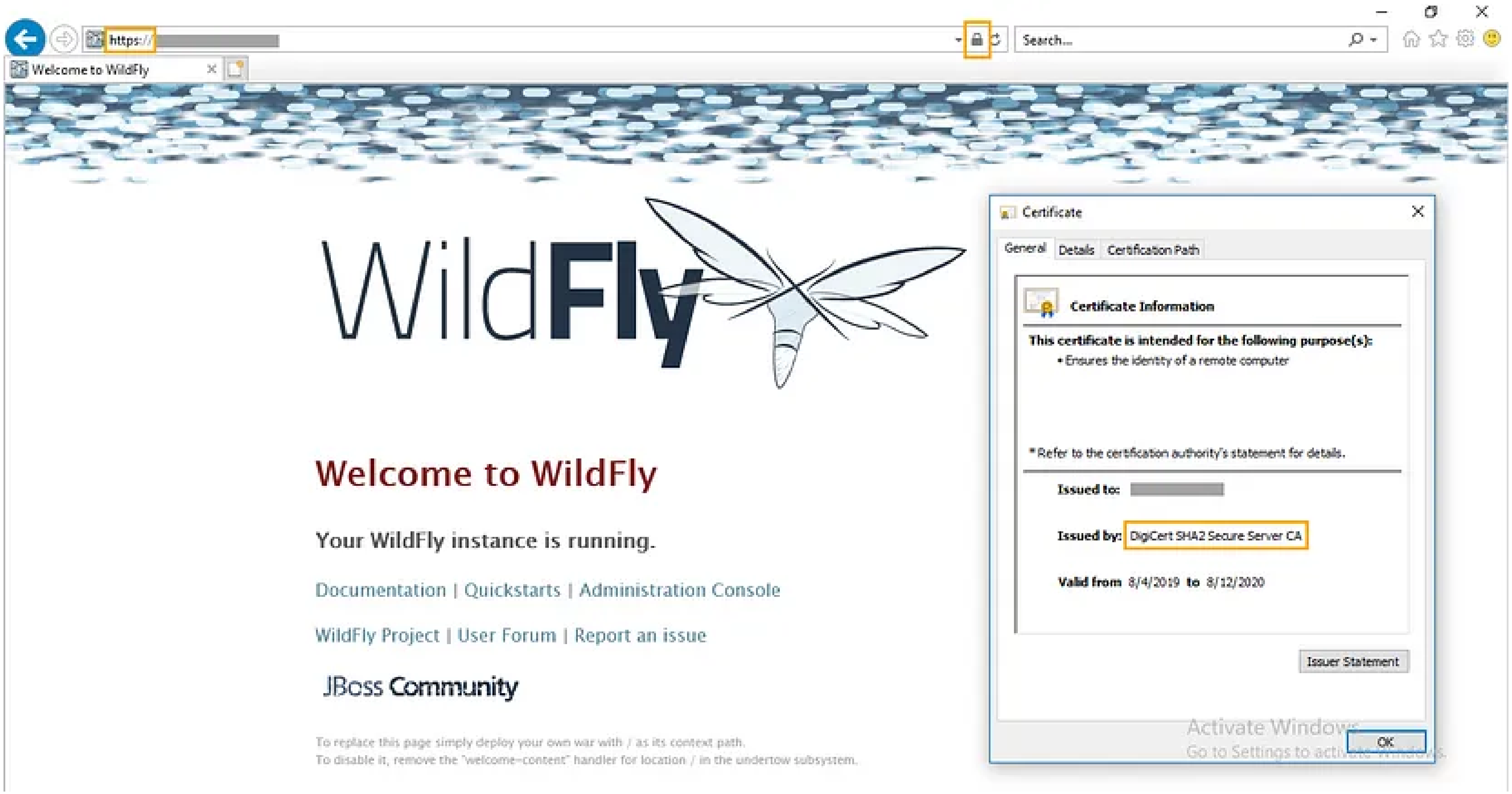
To that run the below commands:

```
$ sudo iptables -t nat -A PREROUTING -p tcp --dport 443 -j REDIRECT --to-ports 8443
```

```
  $ sudo iptables -t nat -A OUTPUT -p tcp --dport 443 -o lo -j REDIRECT --to-port 8443
  $ sudo service iptables save
```

Now go to **https://<your-ssl-registered-domain>:443/** and you can access the Wildlfy server. In addition to this, you will also notice a " **lock**" icon on the right side of your browser's address bar (if you are using the IE browser) as shown below. This means that the browser has validated your SSL certificate and its associated certificate chain and the SSL handshake was successful. The communication between your browser and the server is now HTTP secured.

If you now click the "**lock**" icon (again for IE users), a window pops up and you can see the details of the SSL certificate. As you can see below, the SSL certificate for my domain was issued by the **DigiCert SHA2 Secure Server CA** which is an Intermediary Root CA of **DigiCert**.



Welcome to WildFly

Your WildFly instance is running.

Documentation | Quickstarts | Administration Console

WildFly Project | User Forum | Report an issue

JBoss Community

To replace this page simply deploy your own war with / as its context path.
To disable it, remove the "welcome-content" handler for location / in the undertow subsystem.

Certificate Information

This certificate is intended for the following purpose(s):
- Ensures the identity of a remote computer

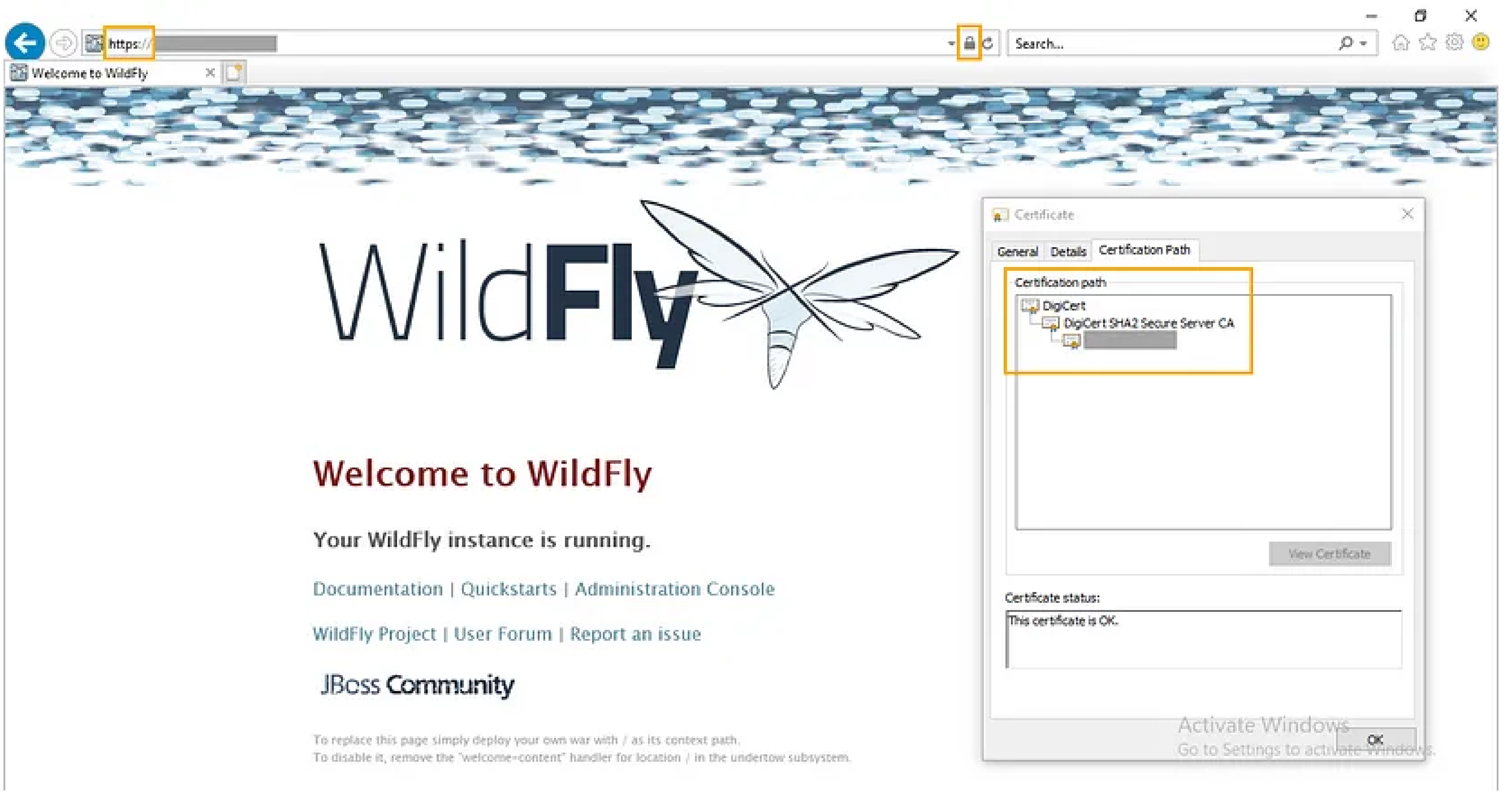
*Refer to the certification authority's statement for details.

Issued to: [redacted]

Issued by: DigiCert SHA2 Secure Server CA

Valid from 8/4/2019 to 8/12/2020

Also, if you click the "**Certification Path**" tab, you can actually see the entire certification chain laid out visually as we discussed above.



Welcome to WildFly

Your WildFly instance is running.

Documentation | Quickstarts | Administration Console

WildFly Project | User Forum | Report an issue

JBoss Community

To replace this page simply deploy your own war with / as its context path.
To disable it, remove the "welcome-content" handler for location / in the undertow subsystem.

Certification path

DigiCert
DigiCert SHA2 Secure Server CA
[redacted]

Certificate status:

This certificate is OK.

This concludes our tutorial on setting up SSL/HTTPS on the JBoss Wildfly server.

***Thanks for reading!!*** *If you enjoyed this article, please clap and leave a comment. Also, check out my other* ***articles*** *as well.*

Here are some useful resources that inspired me to create the content of this article: