# Configuring Couchdb for Inplace 8.0

Joop Ringelberg 18-01-21 Version: 1

## Introduction

Starting with version 8.0, the screens that provide a perspective on contexts are run as ordinary webpages in a browser. InPlace is from now on a Web App[1]. This has many advantages, one of them being able to have multiple windows open side by side. Each page communicates with the same Perspectives Distributed Runtime and this PDR executes in a *service worker*. Standards require PWA's to be served over the secure protocol https and as a consequence, the PDR can now only make https calls.

The PDR stores data in Couchdb, which operates as a local webserver. Consequently, starting with version 8.0, Couchdb must be accessible over the https protocol (otherwise the PDR cannot reach it). This requires some extra configuration.

This situation will change again with coming releases, as we move away from Couchdb to IndexedDB. From then on you will be able to run InPlace without these extra configuration steps.

If you are familiar with concepts like https, encryption and certificates, you can skip the next section to the procedures that follow. However, if you are not, we encourage you to read on.

## Transport Layer Security

A browser uses the hypertext transfer protocol to fetch pages and other resources from a webserver. Websites are run by people or organizations, but the web can be an unsafe place and it has happened that unsuspecting users were deceived into believing they communicate with a party while in reality they were dealing with imposters. To mitigate this risk, many websites now serve their resources over https: the http protocol with added 'transport layer security' (TLS). This serves two purposes:

1. To prove the identity of the parties involved;
2. To protect the information they send to each other from eavesdroppers between their computers.

In practice, a webserver proves its identity by sending a *certificate*, while the end user usually has to sign in with a username and password.

[1] A Progressive Web App, to be precise. However, there is not much progressive to InPlace: it either works or does not work in your browser, there is no notion of graceful degradation.

## Protecting information

Let's focus first on the second purpose, protecting information. This is achieved using cryptography. The particular form of cryptography used in TLS relies on two 'keys': a public one and a private one. The webserver sends the public one to the browser; it is part of the certificate.

Now the imagery of two keys is not very enlightening. It is better to consider the public key to be a kind of *box*, that can only be unlocked by the (private) key. So the server sends a box to the browser; when the user wants to send some sensitive information to the server, the browser puts it into the box, clicks it shut and sends it back. The server can open the box with its key. No one intercepting the box along the way can open it, as they do not have the key. This is the practice of encrypting information to communicate it privately.

But wait! How do we know that the box received by the browser is, indeed, the one sent by the server? If we reckon with interceptors, what if a crook should insert his *own box* in the response sent by the server!? The browser puts the sensitive data in it and gone is the users secret, as the crook intercepts the reply and opens the box with his own (private) key.

So the webserver should prove its identity, or, rather, prove that the public key (box) it ships with its responses is indeed owned by it (the first purpose of TLS). This is where the certificate comes in, and with it the notion of *signing*.

## Proving identity: signing

The King's signature on a banknote should convince people it is genuine. Only the King can create that signature and all can verify that, the King being a well-known public figure! Obviously, long gone are the days that all we needed was that signature, but the principle is clear. This is signing: only one party can put their sign on something and all others have the ability to verify it.

Now if this sounds familiar, you are right. It is just like protecting information, but the other way round:

- Everyone can put something in the box, but only one person can open it;
- One person can sign, but everyone can verify the signature.

The roles are reversed, or, rather, the keys are switched. On protecting information, the (private) key holder sends out the boxes (the public key). On signing, the (public) key holder puts something in a box, and everyone holding the (not so) private key can open it.

Because we can open the box with the King's key, we know for sure that he is the one that put something in it (cause he keeps the unlocked boxes (=the public key) private). So signing requires distributing the (private) key; protecting requires distributing the box (public key).

## Chains and authorities

To sum up where we stand: to protect information flowing from the browser to the server, we encrypt it using the public key sent by the server (the box). To make sure that public key is really owned by the server, it is signed and so becomes a certificate.

But, you may observe, haven't we just pushed the problem a bit further away? For what is checking a certificate other than using a key (the private one, this time, issued by someone who certified the servers' public key)? *So how do we know that key is genuine?*

In our minds eye we can see a chain of certifiers, each certified by someone higher up in the chain. As a matter of fact, this is exactly how it works. But the buck stops somewhere and that is with a certificate we call a *root*, issued by an *authority*.

Who decides who is an (certificate) authority and who not? This is largely the outcome of a historical process. There is now a smallish number of certificate authorities (CA's) that are regarded as trustworthy.

## Root programs and you

You will have accessed countless websites over https and yet never spent a conscious thought on which CA you can trust. So when exactly did this happen? When did *you* decide to trust, say, the certificates signed by VerySign?

Well, you did not, otherwise than by using your browser and the list of CA's trusted by the *Root Program* active on your computer. Such lists are compiled by the suppliers of operating systems, such as Apple and Microsoft[2]. When you use a Mac, you can inspect that list using the KeyChain App (it keeps your passwords safe, but it also holds the list of CA's and certificates to trust).

To sum up: when you access a website protected with TLS, your browser receives its certificate, that holds a public key it will use to encrypt your information prior to sending it back. Before doing so it checks the certificate is signed by an authority that is in the list of authorities kept in the operating system of your computer.

# Configuring https for Couchdb on your computer

The purpose of the procedures outlined below is twofold:

1. To make the Couchdb webserver send out a certificate with responses to requests coming in over https;
2. To make your browser trust that certificate (by adding it to the computers list of trusted certificates).

Only then will InPlace version 8.0 and higher run properly on your computer.

[2] There are a few other Root Programs. Most are by private companies but the Mozilla organization runs a well-known and public Root Program, too.

## Obtaining the certificate

You can create a self-signed certificate if you wish. The *common name* field in it should be `https://localhost:6984`. This is the secure endpoint of Couchdb on your computer.

Alternatively, you may download a certificate for this endpoint signed by Perspect IT from `https://inplace.works/couchdb.pem`. And `https://inplace.works/privkey.pem`. Notice that Perspect IT is not a certificate authority. Neither do you have to trust us; you will have to declare trust in the certificate yourself. You can safely do so, as it only certifies an endpoint on your own computer (Couchdb, that you have installed yourself).

## Storing the certificate

Following the instructions for configuring https options in Couchdb (see below), put the certificate in the directory `/etc/couchdb/cert`. You probably may be able to store the files in another directory. If so, adapt the lines in the ssl section in your local.ini file accordingly (see below). Make sure the directory where you put the files is readable by the Couchdb process!

## Configuring Couchdb

The official Couchdb documentation describes https options in https://docs.couchdb.org/en/latest/config/http.html#https-ssl-tls-options. From that section we've copied the relevant steps. Before carrying them out, stop Couchdb; after finishing, restart again.

> Now, you need to edit CouchDB's configuration, by editing your local.ini file. Here is what you need to do.
>
> Under the [ssl] section, enable HTTPS and set up the newly generated certificates:
>
> ```
> [ssl]
> enable = true
> cert_file = /etc/couchdb/cert/couchdb.pem
> key_file = /etc/couchdb/cert/privkey.pem
> ```

Where to find the Couchdb configuration files is described here: https://docs.couchdb.org/en/stable/config/intro.html#configuration-files.

## Make the browser trust the certificate

If you use Firefox, open the Certificate Manager (Tools > Options > Advanced > Certificates: View Certificates).

If you run Chrome or Safari on Mac OSX, open KeyChain Access. Select the Certificates Category. Drag and drop the couchdb.pem file on the right. Double click it and change the settings to "Always trust".

We have no instructions for Microsoft Windows.

## Apache proxy workaround for Couchb tls problems

It turns out that Couchdb does not add the qualification “Secure” to cookies sent over https. The Chrome browser requires that qualification (together with SameSite=None) before it accepts cookies from third party domains. As a workaround, we skip the TLS in Couchdb (it cannot effectively connect to most browsers, anyway!), handle it in Apache and rewrite the cookies to add Secure to it as well. Below is a working configuration:

```
<VirtualHost *:6984>
  ServerName localhost
  ProxyPass / http://127.0.0.1:5984/
  ProxyPassReverse / http://127.0.0.1:5984/
  SSLEngine on
  SSLCertificateFile "/private/etc/apache2/ssl/127.0.0.1+1.pem"
  SSLCertificateKeyFile "/private/etc/apache2/ssl/127.0.0.1+1-key.pem"
  Header edit Set-Cookie (.*) "$1; Secure"
</VirtualHost>
```