JUXT

Site

Installation Guide

Table of Contents

This is the long-winded installation guide to explain how to bootstrap a Site instance.

This is partly for pedagogical purposes. It is a good idea to learn in depth how Site's builds up its secure foundation. But if you are in a hurry, there are (or will be) quicker 'install packs' and Docker images that provide shortcuts to setting up a Site instance.

1. Requirements

Before you start, you'll need to have the following installed:

- Java 17+ (Site uses new Java APIs such as `java.util.HexFormat`)
- Clojure [https://clojure.org/guides/getting_started]
- Babashka [https://github.com/babashka/babashka]

2. Before you start

All these instructions are for Arch Linux, please amend accordingly for your own platform, if different.

Run each command under your user account (i.e. not as root).

For documentation purposes, we'll assume your Site server will be set up to run on the host `site.test`. If you choose another hostname, or are setting up for production, please amend these instructions accordingly.

3. Preparing your development environment for Site

3.1. Decide on a fully qualified hostname

Site is a web server and is designed to be accessed over the network. Since Site resources are identified by URIs, you should decide on a hostname in a domain that you control.

For the purposes of learning, development and local testing, it is a good idea to use a test hostname, such as `site.test`. These instructions are written assuming that you've chosen `site.test` as your hostname. If you're using your own hostname in a domain you control, amend these instructions accordingly.

3.2. Configure your local DNS or /etc/hosts

To access Site locally for bootstrapping we need to ensure requests to `site.test` are directed to the local service.

Edit /etc/hosts to direct requests for `site.test` to 127.0.0.1.

```
# Static table lookup for hostnames.
# See hosts(5) for details.
127.0.0.1   localhost site.test
```

3.3. mkcert

Install mkcert

```
sudo pacman -Sy mkcert
```

```
mkcert -install
```

```
Created a new local CA
The local CA is now installed in the system trust store!
The local CA is now installed in the Firefox and/or Chrome/Chromium trust store (requires browser restart)!
```

3.4. Install nginx

```
sudo pacman -Sy nginx-mainline
```

3.5. Create private key and cert

Create a new cert for the local development server, and move these into your nginx config directory.

```
mkcert site.test
sudo mv site.test-key.pem site.test.pem /etc/nginx
```

3.6. Configure nginx

Configure nginx. Use `etc/dev/nginx.conf` as a guide to what to configure. You'll need to reference your private key and cert via the `ssl_certificate` and `ssl_certificate_key` parameters.

```
ssl_certificate      site.test.pem;
ssl_certificate_key  site.test-key.pem;
```

You'll need to increase the size of request bodies you can send through nginx, the default is too restrictive.

```
client_max_body_size 16M;
```

You'll also need to configure the details of your the origin server (Site).

```
http {
  ...
  client_max_body_size 16M;
  ...
  server {
    ...
    ssl_certificate site.test.pem;
    ssl_certificate_key site.test-key.pem;
    ...
    location / {
      proxy_pass http://localhost:2021;
      proxy_set_header Host $host;
      proxy_set_header X-Real-IP $remote_addr;
      proxy_set_header X-Forwarded-Proto $scheme;
  }
}
```

Test your final configuration before you start nginx and fix any errors.

```
sudo nginx -t
```

3.7. Start nginx

Start nginx

```
sudo systemctl enable nginx
sudo systemctl start nginx
```

4. Install Site

4.1. Clone this repo

```
$ git clone https://github.com/juxt/site
```

5. Configure the Site service

5.1. Install the configuration file

There's a sample configuration in `etc` you should copy to `$HOME/.config/site/config.edn`.

```
$ mkdir -p $HOME/.config/site
$ cp site/etc/config.edn $HOME/.config/site/config.edn
```

If you're aren't using `site.test` as a hostname, edit the configuration to replace `https://site.test` with the URI that corresponds to the hostname you have chosen,

6. Start the Site server

Start the Site server:

```
$ site/bin/site-server
```

Alternatively, if you're familiar with Clojure development, you can start the server via the `deps.edn` file and simply 'jack-in' with your editor or IDE as normal.

7. Connect to Site's REPL

If you've run Site via your development environment and 'jacked-in' you'll already have a REPL. Proceed to the next step.

If you're running Site with `site/bin/site-server`, you'll need to connect a terminal to Site to access the REPL. You can do this via port `50505`, which is a socket REPL that Site starts by default.

How you connect to this port is up to you. One way is via `ncat`, but you can replace `ncat` with `telnet`, or `netcat`, depending on what's available for your system.

Arch users can install `ncat` by installing the `nmap` package:

```
$ sudo pacman -Sy nmap
```

```
$ ncat localhost 50505
```

Prefix the command with `rlwrap` if you have it installed.

```
$ rlwrap ncat localhost 50505
```

7.1. Introducing Site's REPL

```
Site by JUXT. Copyright (c) 2021, JUXT LTD.
Type :repl/quit to exit

site>
```

There are a few useful Site REPL commands you should be familiar with:

(ls)

List all resources

(ls <pat>)

List all resources whose URIs match the pattern

(evict! <uri> +)

Kill resource(s) across time

(apply evict! (ls))

Start over. (Delete everything in the database!)

8. Configure an authorization server

Site is a Resource Server, responsible for serving resources such as:

- Static web pages and media
- API responses
- GraphQL responses

As a Resource Server, Site is responsible for protecting the resources owned by users from unauthorized access.

Rather than sending credentials, clients (applications) obtain and use access-tokens to access Site's resources.

Site hosts the authorization server by which clients can obtain access tokens.

```
(install-authorization-server! {:name "Authorization Server"})
(install-token-endpoint! {:xt/id "https://site.test/token"})
(register-application! {:name "My Application"})
```

9. Bootstrap remote access to Site

A Site instance is a collection of documents, stored in XTDB.

Like XTDB, Site is schemaless and requires that you define your own documents. However, by included document attributes known to Site (usually in the `juxt.site.alpha` namespace) Site is able to interpret the documents as web or API resources, and serve them over HTTP.

We need to set up sufficient resources in the REPL so that we no longer need to access Site via the REPL.

Secure remote access to Site resources requires an **access token**.

In this section we use the REPL to build up the minimal resources required to acquire an access token which can let us continue setting up the server remotely, without requiring further REPL access.

An access token is granted for a **subject** and an **application**, so we'll need to create those too.

But first, we need to install some preliminary resources into our REPL.

9.1. Install the capability to call actions

Actions are at the heart of Site:

- Actions allow you to read and write to the database.
- Actions are composeable.
- Actions can call external functions, such as lambdas.

- Actions are restricted to authorized users and applications.
- Actions can be grouped into OAuth2 scopes
- Actions can be exposed to the network, via OpenAPI and GraphQL.

9.1.1. Install the `do-action` function

Actions are executed in an XTDB transaction function.

This guarantees consistency, eliminating potential race-conditions. For example, it's important that any revocations to authorization permissions are fully processed if they occur before a call to an action.

The transaction function also records every action call in an **audit-log**, detailing when the action was called, by whom, which entities were affected and, potentially, other details such as the 'business justification'.

We have to first install the transaction function into the database, so that we can start to call the actions we will create.

Install the do-action transaction function

```
(install-do-action-fn!)
```

9.1.2. Create a REPL subject

Actions are performed by subjects.

A **subject** represents an authenticated person, which will include personal data as well as details about their login session (e.g. the device they are using, whether their email address has been verified, whether their login required use of a second factor, etc.).

To call actions from the REPL, we'll install a subject that can only be used from the REPL. The `(me)` function returns a special built-in identifier for the REPL subject. It can't be used remotely.

Add the REPL subject

```
(put! {:xt/id (me)})
```

In future we might allow different users using the same REPL to identify themselves. Of course, REPL users have no restrictions to what they can do, so this is just for audit purposes among trusted users. Access to the REPL must be restricted to a very limited set of authorized users.

9.2. Define the create-action action

We install the `create-action` action. This is the one action that has to be put directly into the database because we don't have a way of creating actions yet!

Install the create-action action

```
(install-create-action!)
```

Permit the create-action action

```
(permit-create-action!)
```

9.3. Define the grant-permission action

Now that we have our `create-action` function installed we can use it to create an action that will grant permissions.

```
(install-grant-permission-action!)
```

Note that this function will return a copy of the **transaction metadata record** for the transaction that created the grant-permission action.

Transaction Metadata Records

Whenever an action is called, the `do-action` transaction function is executed which results in a **transaction metadata record** being created in the database. If the action is denied, or if errors occur when the action is executed, details will be recorded in the transaction metadata.

This allows us later to answer questions as to whether an action was allowed or denied, with an explanation. We will also be able to answer questions regarding the who, when, why and how for each document in the database.

Example 1. A **transaction metadata record**

A copy of the transaction metadata record is returned as a result of the `do-action` function.

```
{:xt/id "urn:site:action-log:134"
 :xtdb.api/tx-id 134
 :juxt.pass.alpha/subject "urn:site:subjects:repl"
 :juxt.pass.alpha/action "https://site.test/actions/create-action"
 :juxt.pass.alpha/purpose nil
 :juxt.pass.alpha/puts ["https://site.test/actions/grant-permission"]
 :juxt.pass.alpha/deletes []}
```

9.3.1. Permit access to the grant permission action

We need to permit our REPL user to call this grant-permission action, and this permission too needs to be put directly into the database since we don't yet have a way of granting permissions!

Granting the REPL user the permission to grant permssions.

```
(permit-grant-permission-action!)
```


9.4. Create a user

To remotely access the system, we'll first have to build a user.

For this example, we'll model a user as a combination of:

- ☐ A **person**,
- ☐ A person's **identity**, as issued from a trusted entity such as an OAuth2 Authorization Server, Identity Provider or equivalent,
- ☐ A **subject** which represents the person's current **session**, including details of how they logged in.

We don't have to model a user this way, and for some applications this might be overly simplistic. Site allows you to model your own users as you see fit, as long as there is something to represent a **subject**.

For the purposes of this example, we'll create a person entity to represent the person.

We'll use the name `Alice` but feel free to replace this with your own personal details.

9.4.1. Create an action to create a person

First, we'll need to create an action which will create our person entity.

Example 2. Creating the create-person action

Install the create-person action

```
(create-action!
{:xt/id "https://site.test/actions/create-person" ①
 :juxt.pass.alpha/scope "write:admin" ②

 :juxt.pass.alpha.malli/args-schema ③
 [:tuple
   [:map
     [:xt/id [:re "https://site.test/people/\\p{Alpha}{2,}"]]
     [:example/type [:= "Person"]]
     [:example/name [:string]]]]

 :juxt.pass.alpha/process ④
 [
  [:juxt.pass.alpha.process/update-in [0] 'merge {:example/type "Person"}]
  [:juxt.pass.alpha.malli/validate]
  [:xtdb.api/put]]

 ::pass/rules ⑤
 '[
   [(allowed? permission subject action resource)
    [permission ::pass/subject subject]]]}
)
```

① You can choose any id here but it's a good idea to keep to a convention

② Actions are grouped into OAuth2 scopes

③ Arguments must conform to this schema

④ The processing pipeline which transforms arguments into XT transaction operations

⑤ An action declares the rules as to who is authorized to call it

9.4.2. Permit the REPL to call the create-person action

Example 3. Adding the permission for the REPL to create a person

```
(grant-permission!
{:xt/id "https://site.test/permissions/repl/create-person"
 ::pass/subject (me)
 ::pass/action #{"https://site.test/actions/create-person"}
 ::pass/purpose nil
 }
)
```

9.4.3. Create 'Alice' with the create-person action

Example 4. Creating 'Alice'

```
(do-action "https://site.test/actions/create-person"
  {:xt/id "https://site.test/people/alice"
  :example/name "Alice"})
```

9.5. Create a subject entity to represent Alice's login

Now we create the **subject** to represent Alice's login.

We do this because we may want our rules to take into account other aspects of Alice's session rather than just the fact that it belongs to Alice. For example, we may want some actions to be denied if Alice is logging in from an insecure location or from a different country.

9.5.1. Create an action to create a subject

We haven't yet created the ability to create subjects, so let's create another action for doing that.

Example 5. Creating an action for creating the subject

```
(create-action!
{:xt/id "https://site.test/actions/create-subject"
 :juxt.pass.alpha/scope "write:admin"

 :juxt.pass.alpha.malli/args-schema
 [:tuple
   [:map
     [:example/type [:= "Subject"]]
     [:example/person [:re "https://site.test/people/\\p{Alpha}{2,}"]]]]

 :juxt.pass.alpha/process ④
 [
  [:juxt.pass.alpha.process/update-in [0] 'merge {:example/type "Subject"}]
  [:juxt.pass.alpha.malli/validate]
  [:xtdb.api/put]]

 ::pass/rules ⑤
 '[
   [(allowed? permission subject action resource)
    [permission ::pass/subject subject]]]})
```

9.5.2. Permit the REPL to call the create-subject action

Example 6. Adding a permission on the create-subject action

```
(grant-permission!
{:xt/id "https://site.test/permissions/repl/create-subject"
 ::pass/subject "urn:site:subjects:repl"
 ::pass/action #{"https://site.test/actions/create-subject"}
 ::pass/purpose nil
 }
)
```

9.5.3. Call the create-subject action

Example 7. Calling the create-subject action

```
(do-action "https://site.test/actions/create-subject"
  {:xt/id "https://site.test/subjects/alice"
   :example/person "https://site.test/people/alice"})
```

9.6. Create an access token

TODO

10. Run the site tool

The site tool is a command-line utility that allows you to remotely administer site.

If you're on MacOS, you will need to install the GNU version of `readlink`. You can do so with brew:

```
brew install coreutils
ln -s /usr/local/bin/readlink /usr/local/bin/readlink
```

We must first get a token that we can use for API access. This process authenticates to the site server using your password.

Here, replace `admin` with your username (or let it default to your OS username)

```
$ site/bin/site get-token -u admin
```

Now we can use the site tool for remote administration. Try the following:

```
$ site/bin/site list-users
```

11. Post installation steps

11.1. Configure the expiry time for tokens

By default, tokens last for an hour. That can sometimes mean they expire during work sessions. You can set the expiry time of new tokens via the REPL.

```
(put! (assoc (e "http://localhost:2021/_site/token")  ::pass/expires-in (* 24 3600)))
```

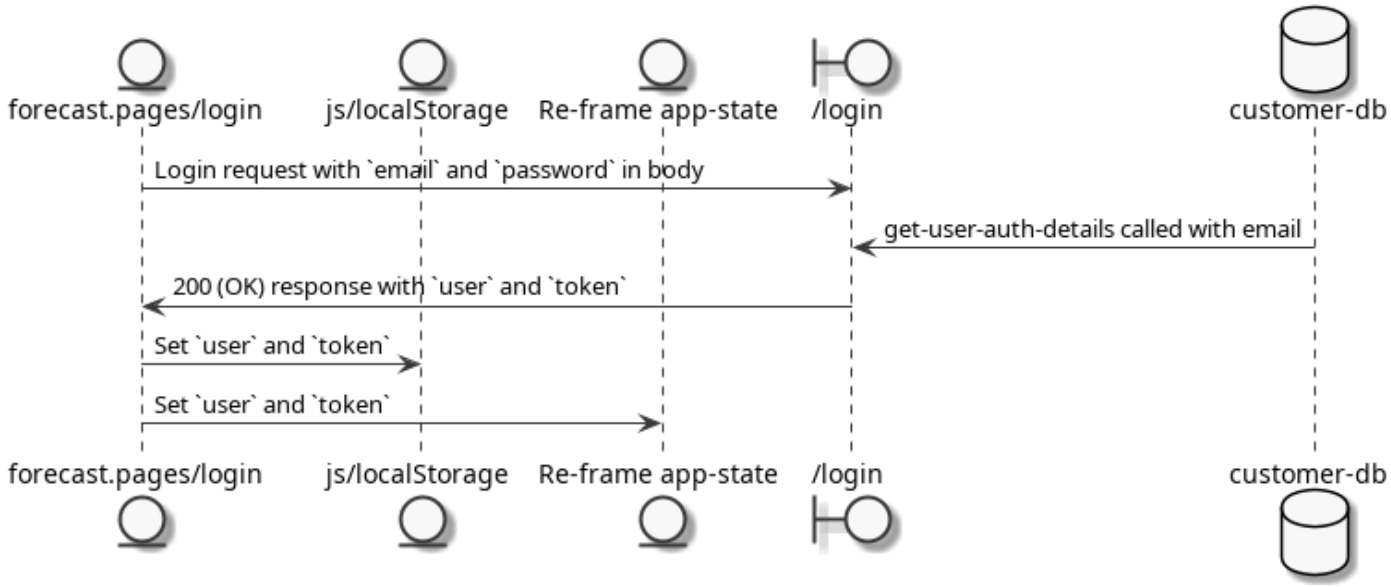

Figure 1. Forecast login sequence and token acquisition