Deployment

Contents

Running Standalone

To create a standalone executable for your application simply run

```
lein uberjar
```

The resulting `jar` can be found in the `target/uberjar` folder. It can be run as follows:

```
java -jar <app>.jar
```

By default the standalone application uses an embedded Immutant server to run the application. However, if you used a profile such as `+jetty` then the alternate server will be used instead. To specify a custom port you need to set the `$PORT` environment variable, eg:

```
export PORT=8080
java -jar <app>.jar
```

Build Tool: lein ▾

Topics

Your First Application
REPL Driven Developme
Application Profiles
HTML Templating
Static Assets
ClojureScript
Routing
RESTful Services
Request types
Response types
Websockets
Middleware
Sessions and Cookies
Input Validation
Security
Component Lifecycle
Database Access
Database Migrations
Logging
Internationalization
Testing
Server Tuning
Environment Variables
- **Deployment**
Useful Libraries
Sample Applications
Upgrading
Clojure Resources

Books

Deploying to WildFly

Without any modifications to your code whatsoever, Immutant-based applications can either run standalone or be deployed to a WildFly app server. The latter requires your app to be packaged in a special war file created by the lein-immutant plugin, e.g.

```
lein immutant war
```

NOTE: The `+war` Luminus profile is incompatible with Immutant apps intended to run on WildFly and should not be used.

More details are available in the official Immutant documentation for WildFly server deployment.

Also, Linode has a guide on how to deploy Luminus application with Immutant and WildFly on Ubuntu 14.04

Deploying to Tomcat

A WAR archive needs to be generated in order to deploy the application to a container such as Apache Tomcat. This is only supported via the lein-uberwar plugin that's included using the `+war` profile.

To enable the `lein-uberwar` plugin manually add the following configuration in the `project.clj` file:

```
:plugins [...
          [lein-uberwar "0.1.0"]]

  :uberwar {:handler <app>.handler/app
            :init <app>.handler/init
            :destroy <app>.handler/destroy
            :name "<app>.war"}
```

In order to create a WAR you can package the application by running:

```
lein uberwar
```

Next, simply copy the resulting `<app>.war` to the `webapps` folder on Tomcat, eg:

```
cp target/uberjar/<app>.war ~/tomcat/webapps/
```

Your app will now be avaliable at the context `/<app>` when Tomcat starts. To deploy the app at root context, simply copy it to `webapp` as `ROOT.war`.

Configuring JNDI Connections

Tomcat may have database configuration specified as a JNDI resource (check JDBC Data Sources section). In this case you need to fetch this data from the Tomcat configuration and not from clojure profiles. Just change these lines in `src/clj/<app>/db/core.clj`:

```
(defstate ^:dynamic *db*
           :start (conman/connect!
                    {:jdbc-url (env :database-url)})
           :stop (conman/disconnect! *db*))
```

to:

```
(def ^:dynamic *db* {:name "java:comp/env/jdbc/EmployeeDB"})
```

(in this example `jdbc/EmployeeDB` is the same name as specified in `context.xml` and `web.xml` in the JNDI HowTo page. Note that this name is case sensitive.)

Configuring JNDI Development Environment

Add the following dependency under the `:project/dev` profile:

```
[directory-naming/naming-java "0.8"]
```

Add the following code in the `user` namespace found in the `env/dev/clj/user.clj` file:

```
(System/setProperty "java.naming.factory.initial"
                    "org.apache.naming.java.javaURLContextFactory")
(System/setProperty "java.naming.factory.url.pkgs"
                    "org.apache.naming")

(doto (new javax.naming.InitialContext)
  (.createSubcontext "java:")
  (.createSubcontext "java:comp")
  (.createSubcontext "java:comp/env")
  (.createSubcontext "java:comp/env/jdbc")
  (.bind "java:comp/env/jdbc/testdb"
         (doto (org.postgresql.ds.PGSimpleDataSource.)
           (.setServerName "localhost")
           (.setDatabaseName "EmployeeDB")
           (.setUser "user")
           (.setPassword "pass"))))
```

The above code will create a JNDI context for the application. Note that you'll have to modify the data source configuration for your particular database configuration.

VPS Deployment

Virtual Private Servers (VPS) such as DigitalOcean provide a cheap hosting option for Clojure applications.

Follow this guide in order to setup your DigitalOcean server. Once the server is created you can install Ubuntu as described here. Finally, install Java on your Ubuntu instance by following these instructions. The instructions below apply for Ubuntu 15.04 and newer.

The most common approach is to run the `uberjar` and front it using Nginx.

Application deployment

In this step, we will deploy your application to the server, and make sure that it is started automatically on boot. We use `systemd` for this. Create a `deploy` user that will run your application:

```
sudo adduser -m deploy
sudo passwd -l deploy
```

Create a directory for your application on the server such as `/var/myapp` then upload your application to the server using `scp`:

```
$ scp myapp.jar user@<domain>:/var/myapp/
```

You should now test that you're able to run the application. Connect to the server using `ssh` and run the application:

```
java -jar /var/myapp/myapp.jar
```

If everything went well, your application now runs locally. The following command will confirm that the applications runs as expected:

```
curl http://127.0.0.1:3000/
```

Your application should also now be accessible on the server at `http://<domain>:3000`. If your application is not accessible make sure that the firewall is configured to allow access to the port. Depending on your VPS provider, you may need to create an access point for the port 3000.

Creating access point on Azure

Creating access point on Amazon EC2

systemd start configuration

Now, let's stop the application instance and create a `systemd` configuration to manage its lifecycle, especially taking care that the application will be launched on server boot. Create the file `/lib/systemd/system/myapp.service` with the following content:

```
[Unit]
Description=My Application
After=network.target

[Service]
WorkingDirectory=/var/myapp
EnvironmentFile=-/var/myapp/env
Environment="DATABASE_URL=jdbc:postgresql://localhost/app?user=app_use
ExecStart=/usr/bin/java -jar /var/myapp/myapp.jar
User=deploy

[Install]
WantedBy=multi-user.target
```

The `WantedBy=` is the target level that this unit is a part of. To find the default run level for your system run:

```
systemctl get-default
```

Note that by default JVM is fairly aggressive about memory usage. If you'd like to reduce the amount of memory used then you can add the following line under the `[Service]` configuration:

```
[Service]
...
_JAVA_OPTIONS="-Xmx256m"
ExecStart=/usr/bin/java -jar /var/myapp/myapp.jar
```

This will limit the maximum amount of memory that the JVM is allowed to use. Now we can tell `systemd` to start the application everytime the system reboots with the following commands:

```
sudo systemctl daemon-reload
sudo systemctl enable myapp.service
```

When the system reboots your application will now start and will be ready to process requests. You may want to test that. Simply reboot your machine, and check the running processes:

```
ps -ef | grep java
```

This should return something like the line below. Pay attention to the `UID` - it should be `deploy`, since running it as `root` would present a significant security risk.

```
deploy     730     1  1 06:45 ?        00:00:42 /usr/bin/java -jar /va
```

Fronting with Nginx

Install Nginx using the following command:

```
$ sudo apt-get install nginx
```

Next, make a backup of the default configuration in `/etc/nginx/sites-available/default` and replace it with a custom configuration file for the application such as:

```
server{
  listen 80 default_server;
  listen [::]:80 default_server ipv6only=on;
  server_name localhost mydomain.com www.mydomain.com;

  access_log /var/log/myapp_access.log;
  error_log /var/log/myapp_error.log;

  location / {
    proxy_pass http://localhost:3000/;
    proxy_set_header Host $http_host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_redirect  off;
  }
}
```

Restart Nginx by running:

```
sudo service nginx restart
```

Then test that the application is available at `http://<domain>`.

Optionally, you can configure Nginx to serve static resources for the application. In order to do that you will need to ensure that all static resources are served using a common prefix such as `static`. Next, upload the `resources/public/static` folder from your application to the server to a location such as `/var/myapp/static` by running the following command from the project folder:

```
scp -r resources/public/static user@<domain>:/var/myapp/static
```

Now add the following additional configuration option under the `server` section of the Ngnix configuration above:

```
location /static/ {
    alias /var/myapp/static/;
  }
```

This will cause Nginx to bypass your application for any requests to `http://<domain>/static` and serve them directly instead.

To enable compression make sure the following settings are present in your `/etc/nginx/nginx.conf`:

```
gzip on;
gzip_disable "msie6";
```

```
gzip_vary on;
gzip_proxied any;
gzip_comp_level 6;
gzip_buffers 16 8k;
gzip_http_version 1.1;
gzip_types text/plain text/css application/json application/x-javascri
```

Setting up SSL

If your site has any user authentication then you will also want to use HTTPS. You will first need to provide a SSL certificate and its key. We'll call these `cert.crt` and `cert.key` respectively.

SETTING UP SSL CERTIFICATE USING LET'S ENCRYPT

The easiest way to setup SSL is to use Certbot and to follow the instructions on the site.

Download the installation tool and generate the certificate using the following commands:

```
git clone https://github.com/certbot/certbot
cd certbot
./certbot-auto certonly --email <you@email.com> -d <yoursite.com> -d <
```

Optionally, setup a Cron job to automatically update the certificate by updating crontab by running as `root`:

```
su
crontab -e
```

Add the following line:

```
0 0 1,15 * * /path-to-certbot/certbot-auto certonly --keep-until-expir
```

Alternatively, you could use Acmetool as a comprehensive solution for keeping certificates up to date.

We'll generate a stronger DHE parameter instead of using OpenSSL's defaults, which include a 1024-bit key for the key-exchange:

```
cd /etc/ssl/certs
openssl dhparam -out dhparam.pem 4096
```

There are two options for handling HTTPS connections. You can either configure the HTTP server in the app itself, or front it with Nginx. We'll look at both approaches below.

App SSL config

You will need to setup Java Keystore to use certificates from the app. This involves running the following commands:

```
openssl pkcs12 -export -in fullchain.pem -inkey privkey.pem -out pkcs.

keytool -importkeystore -deststorepass <PASSWORD_STORE> -destkeypass <
```

If you're using Immutant as your HTTP server (Luminus default), then you have to update your `<app>.core` namespace as follows:

```
(ns <app>.core
  (:require ...)
  (:import
    (java.security KeyStore)
    (java.util TimeZone)
    (org.joda.time DateTimeZone)))

(defn keystore [file pass]
(doto (KeyStore/getInstance "JKS")
  (.load (io/input-stream (io/file file)) (.toCharArray pass))))

(mount/defstate ^{:on-reload :noop} http-server
  :start
  (let [ssl-options (:ssl env)]
    (http/start
      (merge
        env
        {:handler (handler/app)}
        (if ssl-options
          {:port         nil ;disables access on HTTP port
           :ssl-port     (:port ssl-options)
           :keystore     (keystore (:keystore ssl-options) (:keystore-
           :key-password (:keystore-password ssl-options)})))))
  :stop
  (http/stop http-server))
```

The above code will expect the `:ssl` key to be present in the environment. The key should point to a map with the SSL configuration:

```
{:port              443
 :keystore          "/path/to/keystore.jks"
 :keystore-password "changeit"}
```

The application will now be available over HTTPS.

Nginx SSL config

To use Nginx as your SSL proxy you'll want to update the configuration in `/etc/nginx/sites-available/default` as follows:

```
server {
    listen 80;
    return 301 https://$host$request_uri;
}

server {
```

```
    listen 443;
    server_name localhost mydomain.com www.mydomain.com;

    ssl_certificate           /etc/letsencrypt/live/<yoursite.com>/ful
    ssl_certificate_key       /etc/letsencrypt/live/<yoursite.com>/pri

    ssl on;
    ssl_prefer_server_ciphers  on;
    ssl_session_timeout        180m;
    ssl_session_cache  builtin:1000  shared:SSL:10m;
    ssl_protocols  TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers 'AES256+EECDH:AES256+EDH';
    ssl_dhparam /etc/ssl/certs/dhparam.pem;
    add_header Strict-Transport-Security 'max-age=31536000';

    access_log /var/log/myapp_access.log;
    error_log /var/log/myapp_error.log;

     # If you use websocket over https, add below two lines.
    proxy_set_header Upgrade $http_upgrade; ###
    proxy_set_header Connection "Upgrade";   ###

    location / {

      proxy_set_header        Host $host;
      proxy_set_header        X-Real-IP $remote_addr;
      proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_f
      proxy_set_header        X-Forwarded-Proto $scheme;

      # Fix the “It appears that your reverse proxy set up is broken"
      proxy_pass          http://localhost:3000;
      proxy_read_timeout  90;

      proxy_redirect      http://localhost:3000 https://mydomain.com;
    }
}
```

The above will cause Nginx to redirect HTTP requests to HTTPS and use the provided certificate to serve them.

Finally, configure your firewall to only allow access to specified ports by running the following commands:

```
$ sudo ufw allow ssh
$ sudo ufw allow http
$ sudo ufw allow https
$ sudo ufw enable
```

You can test the SSL configuration using the SSL Server Test.

Heroku Deployment

First, make sure you have Git and Heroku toolbelt installed, then simply follow the steps below.

Optionally, test that your application runs locally:

```
heroku local
```

Now, you can initialize your git repo and commit your application:

```
git init
git add .
git commit -m "init"
```

Create your app on Heroku:

```
heroku create
```

Optionally, create a database for the application:

```
heroku addons:create heroku-postgresql
```

The connection settings can be found at your Heroku dashboard under the add-ons for the app.

Deploy the application:

```
git push heroku master
```

Your application should now be deployed to Heroku!

To initialize or update your database:

```
heroku run lein run migrate
```

If the above step uses too much memory for your instance, modify the `start-app` function in your project's `core.clj`:

```
(defn start-app [args]
  (doseq [component (-> args
                        (parse-opts cli-options)
                        mount/start-with-args
                        :started)]
    (log/info component "started"))
  (migrations/migrate ["migrate"] (select-keys env [:database-url]))
  (.addShutdownHook (Runtime/getRuntime) (Thread. stop-app)))
```

This will run migrations every time, keeping the schema up-to-date, while also consuming less resources because the app is already running.

For further instructions see the official documentation.

Enabling nREPL

Luminus comes set up with nREPL, which allows connecting to a REPL on the server. This functionality can useful for debugging as well as hotfixing updates in the running application. To enable nREPL support set the `NREPL_PORT` envrionment variable to the desired port.

```
export NREPL_PORT=7001
```

To test the REPL connection simply run the following command:

```
lein repl :connect 7001
```

You can also connect your favorite IDE to a remote REPL just as you would connect to a local one.

When running on a remote server it is recommended to forward the REPL port to the local machine using SSH:

```
ssh -L 7001:localhost:7001 remotehost
```

Resources

Deploying Your First Clojure App ...From the Shadows provides an indepth guide for Clojure web application deployment strategies.
