Extracted from:

Web Development with Clojure, 2nd Edition

Build Bulletproof Web Apps with Less Code

This PDF file contains pages extracted from *Web Development with Clojure, 2nd Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2016 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Web Development with Clojure

2nd Edition

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

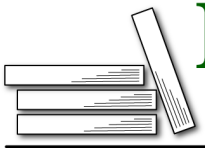
edited by Michael Swaine

Web Development with Clojure, 2nd Edition

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (index)
Candace Cunningham, Molly McBeath (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2016 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-082-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

CHAPTER 6

Writing RESTful Web Services

In the last chapter, we saw how we can leverage ClojureScript on the client to handle the UI state. This allowed us to refactor our server-side code into stateless service operations. However, our services were declared in ad-hoc fashion using an arbitrary convention. While this approach works fine for small applications, it doesn't scale well for situations where we have many service operations and many common concerns, such as authentication, that are shared between them. In this chapter we'll cover how to use a library called compojure-api to provide some structure for our service end points.[1]

Use Compojure-api

Having a well-documented and discoverable service API is key to building a stable and maintainable application. Let's see how Compojure-api allows us to achieve this goal. The Schema library is used to define the Swagger-style RESTful service end points.[2,3] Schema provides us with a number of benefits, such as documentation for the structure of the data, input validation, and optional data coercion. Couple this with the Ring-Swagger library and we can automatically generate an interactive documentation page for our API.[4]

Let's see how this all works in practice by working through a new project. The goal of our project will be to entertain the users with an endless stream of cat pictures. Luckily for us, a public API exists for just such an occasion —the Cat API site.[5] Our app will fetch the cat picture links from there and use these links to display the pictures on the page using ClojureScript.

1. https://github.com/metosin/compojure-api
2. https://github.com/plumatic/schema
3. http://swagger.io/
4. https://github.com/metosin/ring-swagger
5. http://thecatapi.com/docs.html

• Click HERE to purchase this book now. discuss

Let's create the project with the +swagger and the +cljs profiles. The first flag adds the boilerplate for using Compojure-api, and the second enables ClojureScript support out of the box.

```
lein new luminus swagger-service +swagger +cljs
```

The project has a couple of sample routes already set up for us. Let's take a look at these to get a taste for how Compojure-api works. The generated routes can be found in the swagger-service.routes.services namespace. We can see that routes are declared using Compojure-api helpers such as compojure.api.sweet/GET, as opposed to the compojure.core/GET that we used previously. The syntax for these end points is similar to the standard Compojure syntax except that it also requires us to annotate each service operation, as seen here:

```
(GET "/plus" []
     :return       Long
     :query-params [x :- Long, {y :- Long 1}]
     :summary      "x+y with query-parameters. y defaults to 1."
     (ok (+ x y)))
```

The service routes are wrapped using the context macro that sets the base path as /api for all the routes inside it. The macro also specifies a :tags key that contains the metadata for grouping the routes in the generated documentation.

Each service route must declare its return type and the types of its parameters, and it must provide a description of its functionality. We can further see that we must specify where the parameters are found in the request. We specify :query-params for the URL query parameters, :body-params when the parameters are part of the request body, and :path-params when the parameters are part of the request path. Alternatively, we can specify the :body key that points to the description of the request body.

This is seen in the last two /echo routes:

```
(PUT "/echo" []
         :return   [{:hot Boolean}]
         :body     [body [{:hot Boolean}]]
         :summary  "echoes a vector of anonymous hotties"
         (ok body))

(POST "/echo" []
      :return   Thingie
      :body     [thingie Thingie]
      :summary  "echoes a Thingie from json-body"
      (ok thingie))
```

The :body key points to a vector that has the symbol name on the left and the type on the right. The request body is checked against the type specified and

bound to the supplied name. The first route creates an anonymous inline schema definition, while the second uses the schema element that's defined at the top of the namespace.

The :return key specifies the type returned by the function. This can be a simple type, such as Long, or a complex schema, such as the Thingie in the example.

We can also see that the first argument to service-routes is a configuration map. This map specifies the routes for the JSON API, a Swagger UI test page, and the description metadata for the services.

```
{:swagger {:ui "/swagger-ui"
           :spec "/swagger.json"
           :data {:info {:version "1.0.0"
                         :title "Sample API"
                         :description "Sample Services"}}}}
```

Let's start the application and see what the generated documentation looks like by executing lein run in the terminal. Navigate to http://localhost:3000/swagger-ui/index.html, where you should see a page listing the API end points defined in our services namespace.

We can try out the services directly from the page and see how they behave. Note that we're able to call services that use the HTTP POST method without getting the anti-forgery errors we saw in the last chapter. Since anti-forgery protection only makes sense for pages generated by the server, it's not applicable to public API end points. Therefore, the generated service-routes are not wrapped using the wrap-csrf middleware and are exempt from CSRF checks.

swagger-service/src/clj/swagger_service/handler.clj

```
(def app-routes
  (routes
    #'service-routes
    (wrap-routes #'home-routes middleware/wrap-csrf)
    (route/not-found
      (:body
        (error-page {:status 404
                     :title "page not found"}))))))
```

Now that we've seen a few examples of how Compojure-api works, let's go ahead and write a service end point using it.

Creating the API

As the first step, let's see how we can connect to the REST API provided by the Cat API site and extract the content that we need.

Parsing Cat Picture Links

In order to fetch our links, we need to create an HTTP client. Let's use the excellent clj-http library for this task.[6] As always, we start by adding the dependency to our project.clj file.

```
:dependencies [... [clj-http "2.0.0"]]
```

Note that if your application is running, you need to restart it in order for the library to become available.

With the library in place, we can navigate to the swagger-service.routes.services namespace and add the code for reading the links. Let's first reference it in our namespace as client.

```
(:require ...
          [clj-http.client :as client])
```

Let's test getting some data back from the service by writing this function:

```
(defn get-links [link-count]
  (client/get
    (str
      "http://thecatapi.com/api/images/get?format=xml&results_per_page="
      link-count)))
```

The function calls the HTTP GET method on the remote server and requests the results packaged using the XML format. We pass in the number of results to fetch as a parameter. When we call the function in the REPL, we should see something like the following as the result:

```
(get-links 3)
=>
{:status 200
 :headers {"Date" "Sun, 15 Nov 2015 07:00:35 GMT"
           "Server" "Apache"
           "X-Powered-By" "PHP/5.4.45"
           "Connection" "close"
           "Transfer-Encoding" "chunked"
           "Content-Type" "text/xml"}
 :body
 "<?xml version=\"1.0\"?>
  <response>
    <data>
      <images>
        <image>
          <url>http://24.media.tumblr.com/tumblr_m4ikzs7XtT1r6jd7fo1_500.jpg</url>
          <id>egs</id>
```

6. https://github.com/dakrone/clj-http

```
          <source_url>http://thecatapi.com/?id=egs</source_url>
        </image>
        <image>
          <url>http://25.media.tumblr.com/tumblr_m4i1k0nsyd1r3ikrmo1_1280.jpg</url>
          <id>a1c</id>
          <source_url>http://thecatapi.com/?id=a1c</source_url>
        </image>
        <image>
          <url>http://24.media.tumblr.com/tumblr_liqpn3aOHC1qfv1wpo1_500.jpg</url>
          <id>d60</id>
          <source_url>http://thecatapi.com/?id=d60</source_url>
        </image>
      </images>
  </data>
</response>\n"
 :request-time 681
 :trace-redirects
 ["http://thecatapi.com/api/images/get?format=xml&results_per_page=3"]
 :orig-content-encoding nil}
```

As you can see, the result consists of a map representing the HTTP response from the server. The :body key of this map contains the XML describing the links to the cat pictures that we want. Now that we're getting the data from the remote server, we need a way to parse out the links from it.

Clojure provides the clojure.xml namespace for working with XML data. This namespace contains the parse function that can be used to turn an XML input stream into a Clojure data structure. We need to reference the clojure.xml and the clojure.java.io namespaces to create an input stream from the response string and then parse it.

```
(:require ...
          [clojure.java.io :as io]
          [clojure.xml :as xml])
```

With the namespaces referenced, write the parse-xml function that takes the XML string as the input, gets the byte array from the string, wraps it with an input stream, and passes it to the clojure.xml/parse function to extract the data.

swagger-service/src/clj/swagger_service/routes/services.clj

```
(defn parse-xml [xml]
  (-> xml .getBytes io/input-stream xml/parse))
```

We can now update the get-links function to parse the XML result as follows:

```
(defn get-links [link-count]
  (-> "http://thecatapi.com/api/images/get?format=xml&results_per_page="
      (str link-count)
      client/get
      :body
      parse-xml))
```

When we call the function again, we see a Clojure data structure as the result.

```
(get-links 1)
=>
{:tag :response,
 :attrs nil,
 :content
 [{:tag :data,
   :attrs nil,
   :content
   [{:tag :images,
     :attrs nil,
     :content
     [{:tag :image,
       :attrs nil,
       :content
       [{:tag :url,
         :attrs nil,
         :content
         ["http://25.media.tumblr.com/tumblr_m4371fTcUo1qb4lb6o1_500.jpg"]}
        {:tag :id, :attrs nil, :content ["ddk"]}
        {:tag :source_url,
         :attrs nil,
         :content ["http://thecatapi.com/?id=ddk"]}]}]}]}]}
```

All we have to do now is parse out the :url tags from the :image tags in the data. We can accomplish that by writing a few helper functions.

swagger-service/src/clj/swagger_service/routes/services.clj

```
(defn get-first-child [tag xml-node]
  (->> xml-node :content (filter #(= (:tag %) tag)) first))

(defn parse-link [link]
  (->> link (get-first-child :url) :content first))

(defn parse-links [links]
  (->> links
       (get-first-child :data)
       (get-first-child :images)
       :content
       (map parse-link)))
```

While XML format allows for multiple tags in the :content, most of the tags in the structure we're working with only have a single child node. The get-first-child function is used to extract these tags by their name. Once we parse out the collection of links, we can map the parse-link function across them to get the actual URL strings.

We can now update the get-links function to call parse-links to extract the links from the XML structure.

swagger-service/src/clj/swagger_service/routes/services.clj

```
(defn get-links [link-count]
  (-> "http://thecatapi.com/api/images/get?format=xml&results_per_page="
      (str link-count)
      client/get
      :body
      parse-xml
      parse-links))
```

Finally, you might have noticed that the app produces a very noisy log in the console each time we call the client/get function. The log should look something like the following:

```
[2016-02-02 19:12:17,802][DEBUG][org.apache.http.wire]
>> "GET /api/images/get?format=xml&results_per_page=50 HTTP/1.1[\r][\n]"
[2016-02-02 19:12:17,803][DEBUG][org.apache.http.wire]
>> "Connection: close[\r][\n]"
[2016-02-02 19:12:17,804][DEBUG][org.apache.http.wire]
>> "accept-encoding: gzip, deflate[\r][\n]"
[2016-02-02 19:12:17,804][DEBUG][org.apache.http.wire]
>> "Host: thecatapi.com[\r][\n]"
[2016-02-02 19:12:17,804][DEBUG][org.apache.http.wire
 >> "User-Agent: Apache-HttpClient/4.5 (Java/1.8.0_25)[\r][\n]"
[2016-02-02 19:12:17,804][DEBUG][org.apache.http.wire]
>> "[\r][\n]"
```

The reason is that the global logging configuration is set to debug level in the development mode. This causes any libraries we use to log at this level as well. However, we can easily fix this problem by adding an exclusion for org.apache.http in the log configuration. Let's open the env/dev/resources/log4j.properties file and add the following line there:

```
log4j.logger.org.apache.http=INFO
```

When we restart the app, the noisy logs should now be gone. The line says that we would like to configure the logger for the org.apache.http package to use info level rather than debug level. Any time you see the logs get noisy, you can use this method to suppress the logs for the particular package.

Creating the API

All that's left to do is to create a compojure-api route for this operation. At this point we can safely remove the existing sample end points from the namespace. Let's replace these with an end point that accepts the number of links to fetch as the argument and return a collection of link strings as its result. The response will be of type [s/Str].

swagger-service/src/clj/swagger_service/routes/services.clj

```
(defapi service-routes
  {:swagger
   {:ui "/swagger-ui"
    :spec "/swagger.json"
    :data {:info {:title "cat link api"
                  :version "1.0.0"
                  :description "cats api"}
           :tags [{:name "thecatapi", :description "cat's api"}]}}}
  (context "/api" []
    :tags ["thecatapi"]
    (GET "/cat-links" []
      :query-params [link-count :- Long]
      :summary "returns a collection of image links"
      :return [s/Str]
      (ok (get-links link-count)))))
```

Since we're using a GET operation, the input parameter is parsed as a string by default. However, compojure-api provides autocoercion for many common data types, such as UUIDs, integers, longs, and Booleans. Therefore, the argument is coerced automatically to the expected type.

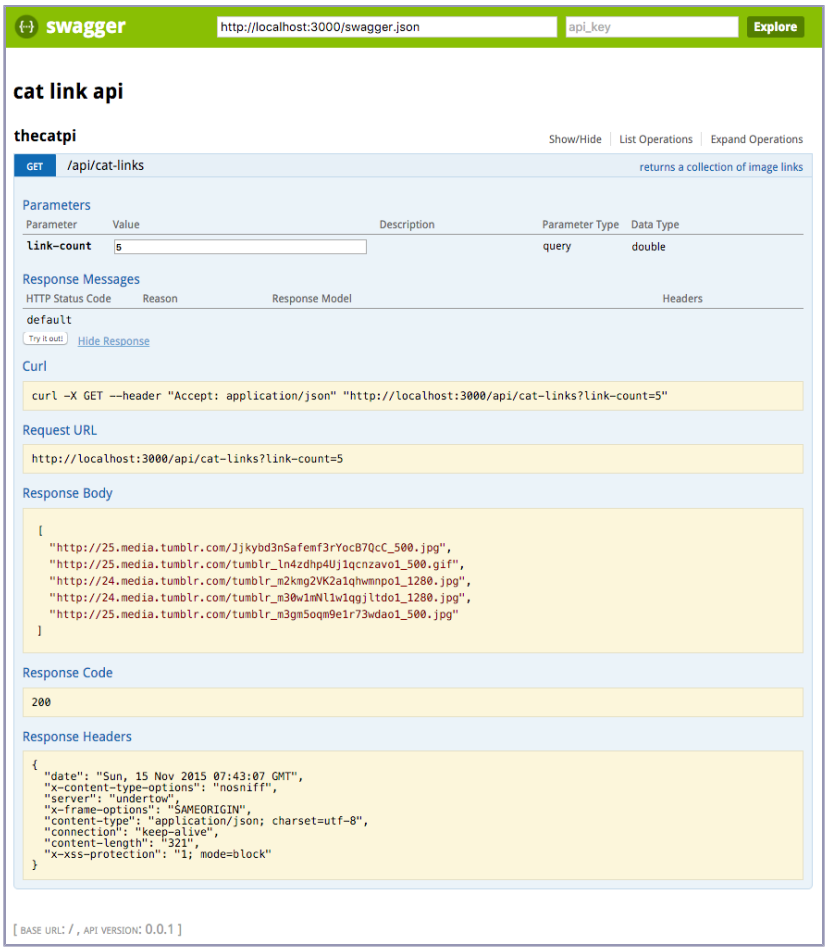
Finally, we can test that our service works as expected by visiting the localhost:3000/swagger-ui/index.html#!/thecatapi/get_api_cat_links page and testing the GET method that we created. We should see something like the figure on page 13.

Next let's take a look at using a ClojureScript client with the Compojure-api service we've just created.

Adding the UI

Now that we have all these exciting links to amazing cat pictures, it would be nice for us to actually see them. Let's navigate to the src/cljs/swagger_service/core.cljs file and add the code to fetch them from the server and display them there. Let's partition the list of URLs that we get from the server and then create pages, each displaying a subset of images.

The lein-cljsbuild plugin we used previously requires us to reload the page each time the sources are recompiled.



When we created our current project, we used the +cljs flag that added ClojureScript support for us. This profile adds a more sophisticated way to compile ClojureScript using the lein-figwheel plugin.[7] This plugin not only compiles the code but also reloads it in the browser as it changes.

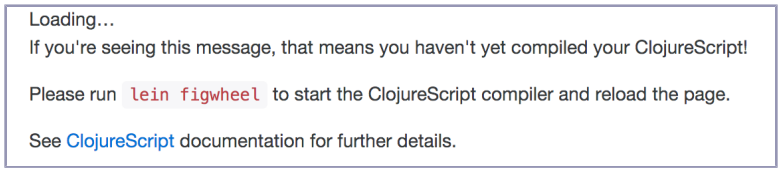
Compiling ClojureScript with Figwheel

With lein-figwheel, the changes are pushed to the browser using a WebSocket and are reflected live without the need to reload the page. Start the server and navigate to http://localhost:3000 once it's ready.

```
lein run
```

7. https://github.com/bhauman/lein-figwheel

Now you should see the following page, stating that you need to run lein figwheel to compile our ClojureScript sources:



When you run the command you should see the following output in the console. The last line in the output tells us that Figwheel is waiting to connect to the application in the browser.

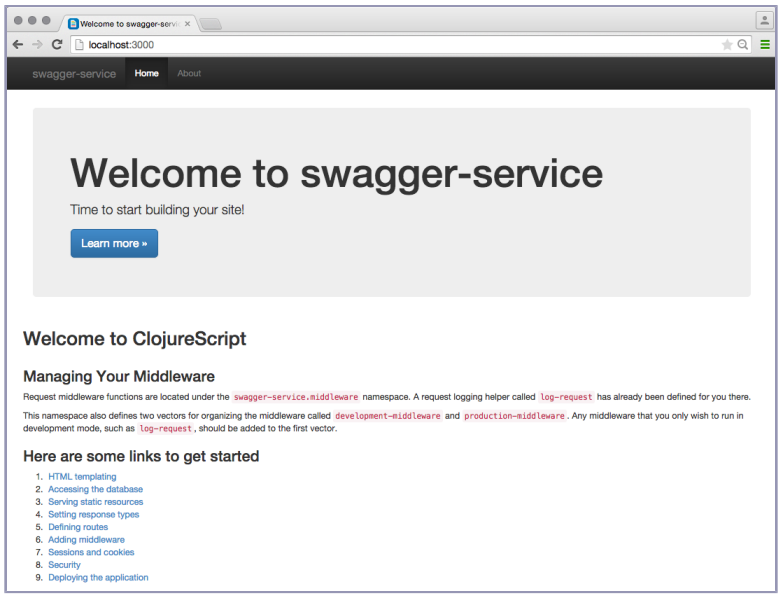
```
lein figwheel
Figwheel: Starting server at http://localhost:3449
Focusing on build ids: app
Compiling "target/cljsbuild/public/js/app.js" from ["src/cljs" "env/dev/cljs"]...
Successfully compiled "target/cljsbuild/public/js/app.js" in 8.719 seconds.
Started Figwheel autobuilder

Launching ClojureScript REPL for build: app
Figwheel Controls:
(stop-autobuild)                ;; stops Figwheel autobuilder
(start-autobuild [id ...])      ;; starts autobuilder focused on optional ids
(switch-to-build id ...)        ;; switches autobuilder to different build
(reset-autobuild)               ;; stops, cleans, and starts autobuilder
(build-once [id ...])           ;; builds source one time
(clean-builds [id ..])          ;; deletes compiled cljs target files
(fig-status)                    ;; displays current state of system
(add-dep [org.om/om "0.8.1"]) ;; add a dependency. very experimental
Switch REPL build focus:
:cljs/quit                      ;; allows you to switch REPL to another build
Docs: (doc function-name-here)
Exit: Control+C or :cljs/quit
Results: Stored in vars *1, *2, *3, *e holds last exception object
Prompt will show when figwheel connects to your application
```

Once you reload the page you should see a navbar and a "Welcome to ClojureScript" message on the page. These elements were generated by the compiled ClojureScript, as seen in the figure on page 15.

We can now navigate to the src/cljs/swagger_service/core.cljs file that contains our swagger-service.core ClojureScript namespace and start editing it. Any changes we make will be reflected live in the browser. For example, let's change the content of the home-page function as follows:

```
(defn home-page []
  [:div
   [:h2 "Welcome to ClojureScript"]
   [:p "live code reloading is fun!"]])
```



Note that any compilation errors or warnings are displayed directly on the page, as seen in the figure on page 16.

At mentioned earlier, Figwheel uses a WebSocket to push the code to the browser. The socket requires additional code to be run when the ClojureScript application starts. This code should only be run in development mode, not in production mode.

In order to automate loading different environments for development and production, the template sets up env/dev and env/prod source paths. The dev path is then included in the :dev profile and the prod path is included in the :uberjar profile in the project.clj file.

The env/dev/cljs/dev.cljs file contains the namespace that's the entry point for our ClojureScript application and has the following contents.

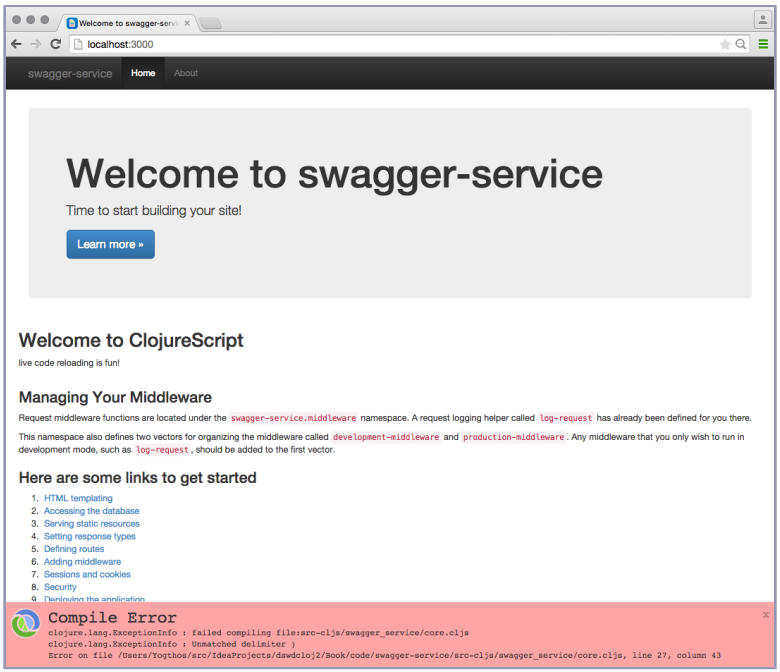
swagger-service/env/dev/cljs/swagger_service/dev.cljs

```
(ns ^:figwheel-no-load swagger-service.app
  (:require [swagger-service.core :as core]
            [figwheel.client :as figwheel :include-macros true]))

(enable-console-print!)

(figwheel/watch-and-reload
  :websocket-url "ws://localhost:3449/figwheel-ws"
  :on-jsload core/mount-components)

(core/init!)
```



It enables console printing, creates a Figwheel WebSocket, and connects the Weasel REPL library to the Clojure nREPL started by the server.[8] Finally, the swagger-service.core/init! function is called. The "prod" entry point disables the console printing functionality and then calls the core/init! function to bootstrap the application.

swagger-service/env/prod/cljs/swagger_service/prod.cljs

```
(ns swagger-service.app
  (:require [swagger-service.core :as core]))

;;ignore println statements in prod
(set! *print-fn* (fn [& _]))

(core/init!)
```

Since the source path is selected based on the profile, no application-specific code needs to be aware of the environment. This approach avoids the need to manually track what parts of the application need to be loaded for development and deployment.

8. https://github.com/tomjakubowski/weasel

Using Figwheel

With Figwheel running, any changes we make in our ClojureScript sources should be automatically updated in the page. Let's replace the existing code from the namespace with the following code:

```
(ns swagger-service.core
  (:require [reagent.core :as reagent :refer [atom]]
            [ajax.core :refer [GET]])
  (:require-macros [secretary.core :refer [defroute]]))

(defn home-page []
  [:div
   [:h1 "TODO: show some cats..."]])

(defn mount-components []
  (reagent/render-component [home-page] (.getElementById js/document "app")))

(defn init! []
  (mount-components))
```

When we check the browser, it should now display the new content without our having to refresh the page. The Figwheel compiler is able to reflect most changes seamlessly. However, we still have to reload the page for certain changes to take effect. For example, when we remove a mounted element such as the navbar, we need to refresh the page for that to take effect.

We're now ready to think about what our UI should look like. We're planning on querying the server for a list of links. We then partition these into groups and show them on the page. Next we need to create an atom to hold the results. Let's create one in our `home-page` component and populate it with some sample links from our API test page and display them using the `img` tag.

```
(defn home-page []
  (let [links
        (atom
         ["http://25.media.tumblr.com/Jjkybd3nSafemf3rYocB7QcC_500.jpg"
          "http://25.media.tumblr.com/tumblr_ln4zdhp4Uj1qcnzavo1_500.gif"
          "http://24.media.tumblr.com/tumblr_m2kmg2VK2a1qhwmnpo1_1280.jpg"
          "http://24.media.tumblr.com/tumblr_m30w1mNl1w1qgjltdo1_1280.jpg"
          "http://25.media.tumblr.com/tumblr_m3gm5oqm9e1r73wdao1_500.jpg"])]
    (fn []
      [:div
       (for [link @links]
         [:img {:src link}])])))
```

As you may recall, we create a local state using the `let` binding and return a function that is called on each subsequent update of this component.

Next, we add a `fetch-links!` function that grabs the list of images from the server. This function accepts an atom along with the number of links to fetch as its

parameters and resets the atom with the links from the server. We can now update our home-page component to fetch the links when it's first called. Then we can see all the links start appearing on the page.

```
(defn fetch-links! [links link-count]
  (GET "/api/cat-links"
       {:params {:link-count link-count}
        :handler #(reset! links %)}))

(defn home-page []
  (let [links
        (atom nil)]
    (fetch-links! links 20)
    (fn []
      [:div
       (for [link @links]
         [:img {:src link}])])))
```

As you can see, that's a lot of links to load all at once. A better user experience would be to partition these into groups and allow the user to navigate these. First, change the fetch-links! function to partition the links into groups of six.

swagger-service/src/cljs/swagger_service/core.cljs

```
(defn fetch-links! [links link-count]
  (GET "/api/cat-links"
       {:params {:link-count link-count}
        :handler #(reset! links (vec (partition-all 6 %)))}))
```

Now write a component that renders a group of links as two rows of images:

swagger-service/src/cljs/swagger_service/core.cljs

```
(defn images [links]
  [:div.text-xs-center
   (for [row (partition-all 3 links)]
     ^{:key row}
     [:div.row
      (for [link row]
        ^{:key link}
        [:div.col-sm-4 [:img {:width 400 :src link}]])])])
```

Let's update the home-page function to track the partition and display it using the component we just wrote:

```
(defn home-page []
  (let [links (atom nil)
        page (atom 0)]
    (fetch-links! links 20)
    (fn []
      [:div
       (when @links
```

```
        [images (@links @page)])])))
```

All that's left is to create a pager component to allow us to navigate the partitions. The pager creates buttons based on the count of partitions and hooks up the logic to navigate back and forth within them. Any time the value of the page atom is changed, it causes the home-page component to be repainted to show the selected partition.

swagger-service/src/cljs/swagger_service/core.cljs

```
(defn forward [i pages]
  (if (< i (dec pages)) (inc i) i))

(defn back [i]
  (if (pos? i) (dec i) i))

(defn nav-link [page i]
  [:li.page-item>a.page-link.btn.btn-primary
   {:on-click #(reset! page i)
    :class    (when (= i @page) "active")}
   [:span i]])

(defn pager [pages page]
  (when (> pages 1)
    (into
      [:div.text-xs-center>ul.pagination.pagination-lg]
      (concat
        [[:li.page-item>a.page-link.btn
          {:on-click #(swap! page back pages)
           :class    (when (= @page 0) "disabled")}
          [:span "«"]]]
        (map (partial nav-link page) (range pages))
        [[:li.page-item>a.page-link.btn
          {:on-click #(swap! page forward pages)
           :class    (when (= @page (dec pages)) "disabled")}
          [:span "»"]]]))))
```

Note that the pager function uses unicode characters for the forward and backward arrows. Alternatively, we could use HTML codes, as follows:

```
(ns swagger-service.core
  (:require ...
            [goog.string :as gs]))

...

(defn pager [pages page]
  (when (> pages 1)
    (into
      [:div.text-xs-center>ul.pagination.pagination-lg]
      (concat
        [[:li.page-item>a.page-link.btn
          {:on-click #(swap! page back pages)
```

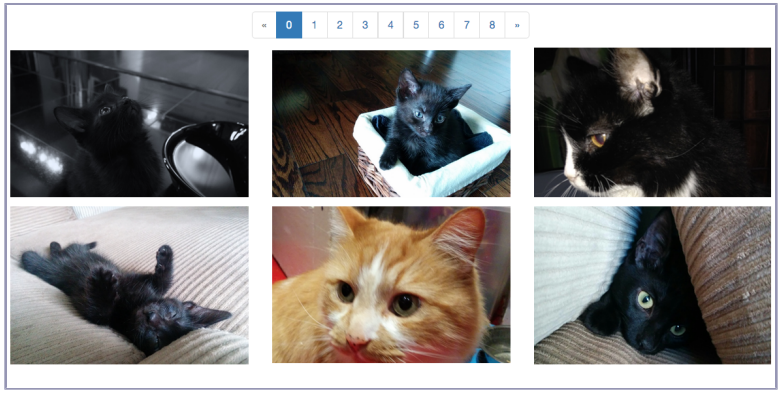
```
          :class    (when (= @page 0) "disabled")}
         [:span (gs/unescapeEntities "&laquo;")]]]
       (map (partial nav-link page) (range pages))
       [[:li.page-item>a.page-link.btn
         {:on-click #(swap! page forward pages)
          :class    (when (= @page (dec pages)) "disabled")}
         [:span (gs/unescapeEntities "&raquo;")]]])))))
```

Finally, let's update the `home-page` component to add the pager:

swagger-service/src/cljs/swagger_service/core.cljs

```
(defn home-page []
  (let [page  (atom 0)
        links (atom nil)]
    (fetch-links! links 50)
    (fn []
      (if (not-empty @links)
        [:div.container>div.row>div.col-md-12
         [pager (count @links) page]
         [images (@links @page)]]
        [:div "Standby for cats!"])))))
```

That's all there is to it. We're now fetching data from the server, partitioning it into groups, and providing a way to navigate these partitions, all in under a hundred lines of code.



What You've Learned

We now have a way to organize our service end points in a structured way, and we have a much better development story when it comes to ClojureScript compilation. In the next chapter we'll take a deeper look at connecting to and working with databases.