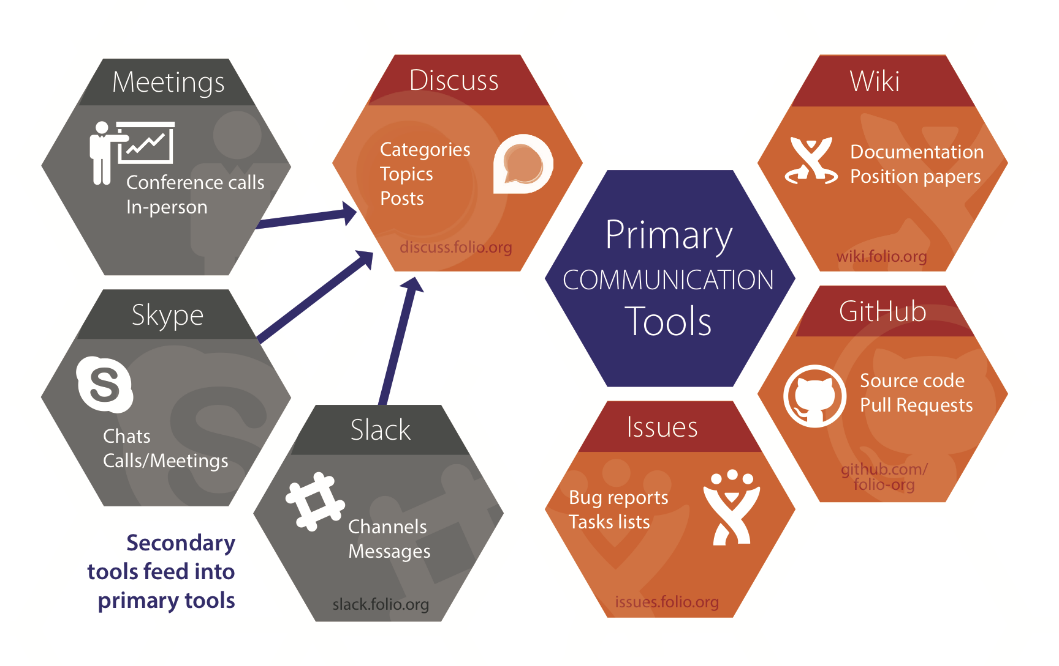
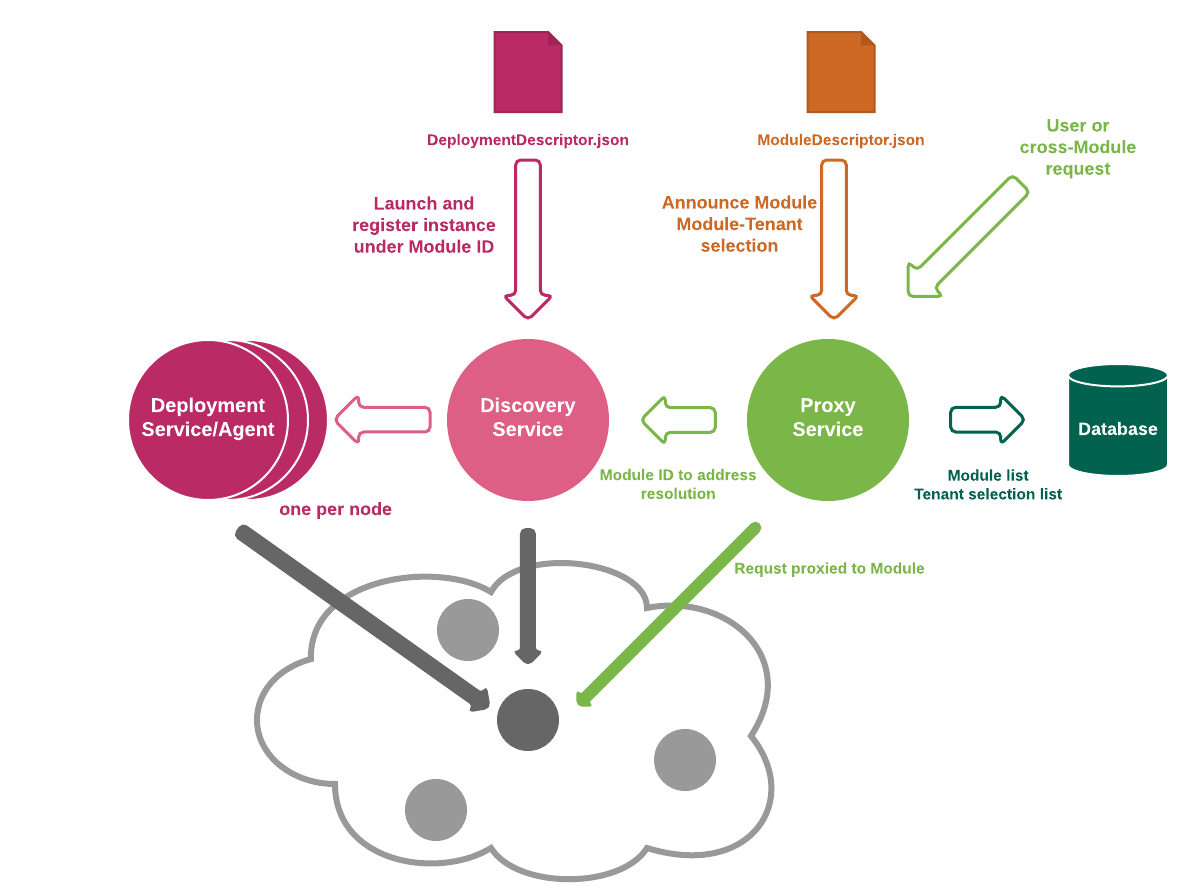
|  |
| --- |
| **VALA Tech Camp 2017 – FOLIO | Wednesday 12 July, 2017, 1p.m. to 5 p.m. |William Angliss Conference Centre, Melbourne** |

Workshop Program

**FOLIO Communication Spaces**

**FOLIO Developer Curriculum**

Workshop Facilitators

**Richard Burkitt: EBSCO Director of SaaS Innovation – Europe, Australia, New Zealand & South Africa**

Richard has worked in the information industry for 15+ years for a number of high profile organisations. He’s held a variety of positions, split between commercial and technical perspectives. In his current role with EBSCO Richard has responsibilities for EBSCO Discovery Service and FOLIO - the Open Source, community-driven Library Services Platform, in Europe, Australia, New Zealand and South Africa. Prior to joining EBSCO Richard led the Sales Engineering team in Europe, Middle East and Africa for Serials Solutions/ProQuest for 8 years. Richard’s motivations include the use of technology in education and in particular working with library patron and staff services to enhance resource use and streamline workflows within the physical and virtual library. Richard lives in Yorkshire, UK.

**Alvet Miranda: EBSCO Library Services Engineering Manager – South West Asia, Oceania, Africa**

Alvet lives in Melbourne and works as a Library Services Engineering Manager with EBSCO. He has 15+ years’ experience developing software solutions for a wide range of industries and businesses. He holds a BSc. in Software Engineering, Master of Business Administration and Master of Applied Finance including over a dozen certifications such as Project Management Professional (PMP) and Agile Certified Practitioner. He enjoys creating beautiful software and when he is not writing code or travelling to see customers, he does research for his thesis as a doctoral candidate at the Victoria University in Australia.

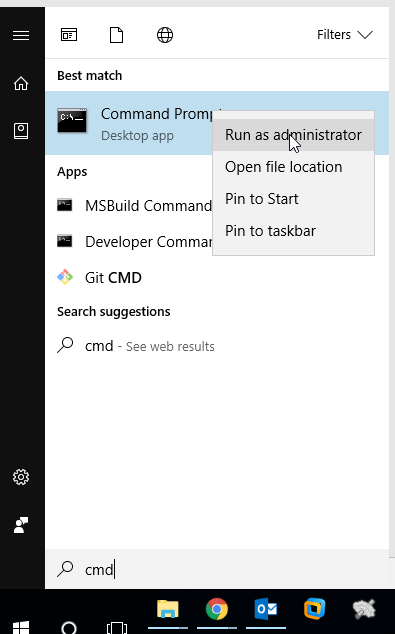
FOLIO Communication Spaces

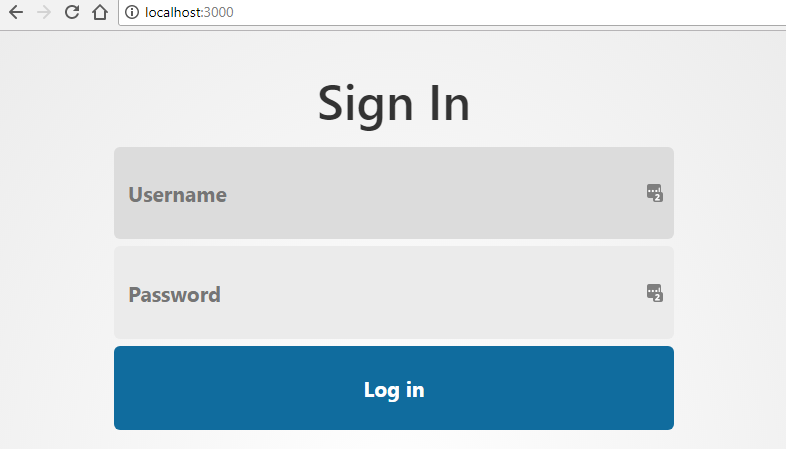
This section will guide you to setup the stable instance of FOLIO located at <https://app.vagrantup.com/folio/boxes/stable>

**Pre-requisites:**

1. Install Oracle virtual box by visiting <https://www.virtualbox.org/wiki/Downloads>
2. Install vagrant by visiting <https://www.vagrantup.com/downloads.html>

**Run FOLIO**

1. Create a folder called folio-stable on your computer. Preferably at C:/folio-stable
2. Download folio-stable.vbox from a LAN address (see Slack) or USB and place in the folio-stable directory.
3. Download Vagrantfile from a LAN address (see Slack) or USB and place in the folio-stable directory. (If not at C:/folio-stable) Open Vagrantfile in Notepad and update the path of the folio-stable.vbox
4. Run Command prompt as administrator (see screenshot)
5. Type cd C:\folio-stable
6. Type vagrant up
7. Wait for the process to complete
8. Open Browser and go to <http://localhost:3000>
9. Login using
   * Username: diku\_admin
   * Password: admin



**Remove FOLIO**

1. Run Command prompt as administrator (see screenshot)
2. Type cd C:\folio-stable
3. Type vagrant destroy
4. Delete the folder C:\folio-stable
5. Uninstall vagrant
6. Uninstall virtual box

FOLIO Developer Curriculum

http://dev.folio.org/curriculum/

**Getting Started with FOLIO for Developers**

00: Introduction  
01: Clone, Build, and Explore Okapi  
02: Initialize Okapi from the command line  
03: Enable Okapi Authentication  
04: Deploy test Stripes module  
05: Set up the Okapi Users app

# 00: Introduction to FOLIO Developer Curriculum Outline

This is an outline of the tutorial for the VALATechCamp workshop developers.

## Goals

* Set up a running instance of the Stripes Development UI Server
* Set up a running instance of Okapi Gateway
* Demonstrate how to deploy an Okapi Module to a tenant in Okapi Gateway and the Stripes UI Server
* Understand how Okapi Gateway routes requests to modules

## System Requirements

There are two choices: either running the Stripes Development UI Server and the Okapi Gateway directly on a developer’s machine (“on-machine”) or running Stripes and Okapi in a VirtualBox guest. An Ansible playbook with appropriate roles is used to create the VirtualBox guest, and can also be used to automatically build a developer’s environment (making the playbook target localhost).

* MacOS 10.? or higher (On-machine or VirtualBox)
* Windows 10 or higher (VirtualBox required)
* Linux (On-machine or VirtualBox)

## Prerequisites

Before attending the workshop, participants must meet these requirements. When in doubt, using the VirtualBox guest machine is recommended.

### VirtualBox guest

* [VirtualBox 5.1 or higher](https://www.virtualbox.org/wiki/Downloads)
* [Vagrant 1.9.1 or higher](https://www.vagrantup.com/downloads.html)
* [Ssh client (Git recommended) – for developers on windows only](https://git-scm.com/)

To download the VirtualBox guest:

1. Make a clean directory and change into it: mkdir folio-curriculum && cd folio-curriculum
2. ~~Set up the Vagrantfile: vagrant init --minimal folio/curriculum~~

|  |
| --- |
| **NOTE: This section is updated for the VALATechCamp17** - Download folio-curriculum.vbox from a LAN address (see Slack) or USB and place in the folio-curriculum directory. - Download folio-valatechcamp17-master.zip from a LAN address (see Slack) or from github alvetm/folio-valatechcamp17 - Extract the contents of folio-valatechcamp17-master.zip to folio-curriculum and move up the files from nested directories if required. - Open the file Vagrantfile and update the path of the folio-curriculum.vbox - Go into folio-tutorial-working-files and extract the files mod-users.zip, okapi.zip and stripes-tutorial-platform.zip - Open three terminals and change directory to folio-curriculum. Windows developers need to run command prompt as administrator. |

1. Launch the VirtualBox guest: vagrant up
2. Connect to the VirtualBox guest: vagrant ssh

|  |
| --- |
| **NOTE: This section is updated for the VALATechCamp17** - Run vagrant ssh in the other 2 terminals. Windows developers can run vagrantssh.bat if there is a ssh client error. - Change to directory /vagrant/folio-tutorial-working-files  The folio-tutorial-working-files contains supporting files to run the commands specified in this tutorial. When directed type in ./<No.> and press tab. Then press enter to execute the command. E.g. in Terminal 1 – **type ./02 then press tab then enter.**  *Run chmod -R +x \*.sh if .sh files don’t execute.*  **Type ./03, tab, enter**  Copy the content of stripes-tutorial-platform\standalone\_package and paste it into stripes-tutorial-platform **In terminal 2 type ./29, tab, enter. ./30, tab, enter**  After execution in terminal 1 is complete; go to terminal 3 and **type ./32, tab, enter. Then type ./33, tab, enter** |

In subsequent lessons, the command lines are executed within the VirtualBox guest. Be sure you are connected to the VirtualBox guest (from the host computer: vagrant ssh) before running the commands.

Other instructions and commands that are specific to the VirtualBox guest mode of using the tutorial are noted using this style of information box.

If launching Vagrant from a Windows Command Prompt, be sure to use Run As Administrator… when opening the Command Prompt itself (cmd.exe). If you are seeing the error “EPROTO: protocol error, symlink”, the likely cause is that Vagrant was not launched with administrator privileges. See issue [STRIPES-344](https://issues.folio.org/browse/STRIPES-344) for details.

## Set FOLIO\_ROOT Variable for Lessons

Each lesson assumes the existence of a $FOLIO\_ROOT shell variable. This variable holds the path to a directory where the components of the Okapi Server and the Stripes Development UI Server are located.

If using the VirtualBox guest setup, it is recommended to first cd /vagrant before creating the empty directory. Doing so makes the Okapi and Stripes files available from the host operating system in the same directory the Vagrantfile file is located.

The first command below connects from the VirtualBox host to the VirtualBox guest. The second command changes the working directory to the shared vagrantdirectory.

$ vagrant ssh  
$ cd /vagrant

Create the directory, enter the directory, and save the directory location using these commands:

$ mkdir folio-tutorial-working-files  
$ cd folio-tutorial-working-files  
$ export FOLIO\_ROOT**=**`pwd`

# 01: Clone, Build, and Explore Okapi

## Clone and Install Okapi

**02-gitCloneOkapi.sh (Terminal 1)**  
$ cd $FOLIO\_ROOT  
$ git clone --recursive <https://github.com/folio-org/okapi.git> Receiving objects: 100% **(**13316/13316**)**, 2.34 MiB | 655.00 KiB/s, **done**.  
 Resolving deltas: 100% **(**6813/6813**)**, **done**.  
 Checking connectivity... **done**.  
 Submodule 'okapi-core/src/main/raml/raml-util' **(**https://github.com/folio-org/raml.git**)** registered **for** path 'okapi-core/src/main/raml/raml-util'  
 Cloning into 'okapi-core/src/main/raml/raml-util'...  
 remote: Counting objects: 636, **done**.  
 remote: Compressing objects: 100% **(**80/80**)**, **done**.  
 remote: Total 636 **(**delta 61**)**, reused 99 **(**delta 39**)**, pack-reused 511  
 Receiving objects: 100% **(**636/636**)**, 113.96 KiB | 126.00 KiB/s, **done**.  
 Resolving deltas: 100% **(**357/357**)**, **done**.  
 Checking connectivity... **done**.  
 Submodule path 'okapi-core/src/main/raml/raml-util': checked out '4d0e256193190f62a4154fc5eebb3ce8cddd21e2'

The first time Okapi is installed will take several minutes as various JAR files are downloaded from the Maven repository. Subsequent installs will not take as long.

**03-mvnInstallOkapi.sh (Executed)**  
$ cd okapi  
$ mvn install  
 **[**...]  
 **[**INFO] ------------------------------------------------------------------------  
 **[**INFO] Reactor Summary:  
 **[**INFO]  
 **[**INFO] okapi .............................................. SUCCESS **[** 10.832 s]  
 **[**INFO] okapi-common ....................................... SUCCESS **[**01:28 min]  
 **[**INFO] okapi-test-module .................................. SUCCESS **[** 12.656 s]  
 **[**INFO] okapi-test-auth-module ............................. SUCCESS **[** 1.374 s]  
 **[**INFO] okapi-test-header-module ........................... SUCCESS **[** 1.431 s]  
 **[**INFO] okapi-core ......................................... SUCCESS **[**02:15 min]  
 **[**INFO] ------------------------------------------------------------------------  
 **[**INFO] BUILD SUCCESS  
 **[**INFO] ------------------------------------------------------------------------  
 **[**INFO] Total time: 04:21 min  
 **[**INFO] Finished at: 2017-05-23T20:23:32+00:00  
 **[**INFO] Final Memory: 38M/264M  
 **[**INFO] ------------------------------------------------------------------------

## Interact with the test modules as if you are the Okapi Gateway

The mvn install command builds okapi-core (the Okapi Gateway server) and okapi-common (utilities used by both the gateway server and by modules) along with three simple test Okapi Modules. Before starting the Okapi Gateway, we are going to look at one of the three test Okapi Modules and interact with them as if we are the Okapi Gateway. Okapi is an implementation of the “API Gateway” microservices pattern. As such, the Okapi Gateway accepts RESTful requests from clients and routes them through a series of RESTful interfaces (“Okapi Modules”) to build a response that is ultimately returned to the client. (For more details about the Okapi architecture, see the [Okapi Guide and Reference](https://github.com/folio-org/okapi/blob/master/doc/guide.md).)

To see what this interaction between okapi-core and Okapi Modules looks like, let’s start an Okapi Module and send it some requests via curl.

### Interact with Okapi-test-module

The Okapi-test-module is a very simple Okapi Module. If you make a GET request to it, it will reply “It works”. If you POST something to it, it will reply with “Hello” followed by whatever you posted. First we start the test module:

**04-RunOkapiTestModule.sh (Terminal 1)**  
$ cd $FOLIO\_ROOT/okapi  
$ java -jar okapi-test-module/target/okapi-test-module-fat.jar  
 13:53:00 INFO MainVerticle Starting okapi-test-module 42510@Walkabout.lan on port 8080  
 13:53:00 INFO ertxIsolatedDeployer Succeeded **in** deploying verticle

With the Okapi-test-module now listening on port 8080, in another terminal window send a simple curl command as shown below.When using the VirtualBox method, you will need to open a new terminal window on your host computer, change the working directory to the location of the Vagrantfile, and use the vagrant ssh command to connect from the host computer to the guest. Note that the -i command line option tells curl to output the response headers in addition to the response body, and the -w '\n' option adds a newline to the end of the response body to ensure the shell prompt starts on a new line.

**05-CallTestb.sh (Terminal 2)**  
$ curl -i -w '\n' <http://localhost:8080/testb> HTTP/1.1 200 OK  
 Content-Type: text/plain  
 Content-Length: 8  
 It works

Next make a HTTP POST request (using -x POST) and send the string Testing Okapi (using the -d command line option):

**06-PostToTestb.sh (Terminal 2)**  
$ curl -i -w '\n' -X POST -d "Testing Okapi" <http://localhost:8080/testb> HTTP/1.1 200 OK  
 Content-Type: text/plain  
 Transfer-Encoding: chunked  
 Hello Testing Okapi

Okapi modules would typically send and receive JSON content bodies, but in these examples simple strings are sent and returned. Leave the Okapi Gateway running (as started by the java -jar okapi-test-module/target/okapi-test-module-fat.jar command above) for the next lesson section below.

### Interact with Okapi-test-module with headers

As a RESTful interface, the Okapi Gateway communicates key data to Okapi Modules and to the client using HTTP headers. For example: as Okapi Modules are chained together by the Okapi Gateway, a module may want to signal to the gateway that it encountered an exception and must interrupt the chain. For example, send an HTTP GET request with an X-my-header: blah header (using the -H command line argument):

**07-TestbMyHeader.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET -H 'X-my-header: blah' <http://localhost:8080/testb> HTTP/1.1 200 OK  
 Content-Type: text/plain  
 Content-Length: 12  
 It worksblah

This example appends the contents of the X-my-header to the response body. If we add an ‘X-stop-here’ header, the module returns the X-Okapi-Stop header (which would trigger the exception handling in the Okapi gateway):

**08-TestbStopHeader.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET -H 'X-my-header: blah' \  
 -H 'X-stop-here: because I said so.' \  
 <http://localhost:8080/testb> HTTP/1.1 200 OK  
 Content-Type: text/plain  
 X-Okapi-Stop: because I said so.  
 Content-Length: 12  
 It worksblah

(Note that there is not an X-stop-here request header defined in Okapi. This is a header specific to the Okapi-test-module that forces the return of the Okapi-defined X-Okapi-Stop response header.) The corresponding Okapi Module code that is handling this interaction can be found in the [okapi-test-module/…/MainVerticle.java](https://github.com/folio-org/okapi/blob/master/okapi-test-module/src/main/java/org/folio/okapi/sample/MainVerticle.java) file.  
Return to the terminal window with Okapi-test-module running and press Control-C to exit it.

# 02: Initialize Okapi from the command line

## Start the Okapi Gateway

In a terminal window, start the Okapi Gateway service.

When using the VirtualBox method, you will need to open a terminal window on your host computer, change the working directory to the location of the Vagrantfile, and use the vagrant ssh command to connect from the host computer to the guest.

**09-RunOkapi.sh (Terminal 1 – hit Ctrl+C then run)**  
$ cd $FOLIO\_ROOT/okapi  
$ java -Dloglevel**=**DEBUG -jar okapi-core/target/okapi-core-fat.jar dev  
 12:08:11 INFO MainVerticle git: git@github.com:folio-org/okapi 225c9c1e03c29459da430f93110abb30378e1394  
 12:08:11 INFO MainVerticle clusterManager not **in** use  
 12:08:11 INFO MainVerticle Proxy using inmemory storage  
 12:08:12 WARN Storage Storage.resetDatabases: NORMAL  
 12:08:12 INFO TenantWebService All tenants deployed  
 12:08:12 INFO MainVerticle API Gateway started PID 64161@Walkabout.lan. Listening on port 9130

The dev parameter starts the Okapi Gateway in ‘development mode’ (a known, clean state without any modules or tenants defined). The Okapi Gateway is using an in-memory database (a built-in PostgreSQL database can be specified by adding -Dstorage=postgres before the -jar parameter). We are going to run the Okapi Gateway with debugging turned on so you can see the effect of the requests passing through the gateway. The last line of output tells us that the Okapi Gateway is running on port 9130.

Open up a second terminal window (noting that if you are VagrantBox method you will need to open a new terminal on your host and use the vagrant ssh command), then use these two curl commands to list the Okapi Modules and tenants known to the gateway:

**10-ListAllModulesOkapi.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET <http://localhost:9130/_/proxy/modules>  
 HTTP/1.1 200 OK  
 Content-Type: application/json  
 Content-Length: 3  
 **[** **]**

**11-ListAllTenantsOkapi.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET <http://localhost:9130/_/proxy/tenants>  
 HTTP/1.1 200 OK  
 Content-Type: application/json  
 Content-Length: 3  
 **[** **]**

Note that in both cases what was returned from the gateway are empty JSON lists, meaning that the newly initialized Okapi Gateway has no configured modules or tenants.

Paths starting with /\_/ are core Okapi Gateway services. /\_/proxy is one core service; it is used to (//TODO define this). Another core service is /\_/discovery; it is used to interact with nodes in the Okapi cluster. More details about these core services can be found in the [Okapi Guide and Reference](https://github.com/folio-org/okapi/blob/master/doc/guide.md). In the next section we will use these two core services to register a module and a tenant.

## Defining the okapi-test-module within the Okapi Gateway

Defining a module with the Okapi Gateway occurs in three steps: Registering, Deploying, and Configuring the proxy.

### Registering okapi-test-module

To tell Okapi that we want to use the okapi-test-module, we create a JSON structure of a “Module Descriptor” and POST it to Okapi. The command below creates a Module Descriptor JSON structure. (Leave the Okapi Gateway running in one window and execute this command in another window.)

$ cd $FOLIO\_ROOT  
$ cat okapi-proxy-test-basic.json

This is a module descriptor for test-module (the first id key-value pair). It provides two interfaces: test-basic and \_tenant. The test-basic interface is how clients will communicate with the Okapi Module (as we did directly in the previous lesson). Interfaces beginning with an underscore (such as \_tenant) are reserved system interfaces; in this case, an interface that is called when a module is enabled or disabled for a tenant. The handlingdictionary tells the gateway:

* which HTTP methods are expected
* the path registered in the gateway for which this interface will receive requests
* the type of response provided by this module interface
* permissions required (enforced by the Okapi Gateway)
* permissions desired (a list of requester’s permissions that the business logic of this module needs to know about) The final entry, launchDescriptor, tells the gateway how to start the Okapi Module.

To deploy the module, this JSON is POSTed to the /\_/proxy/modules core service:

**12-RegisterOkapi-proxy-test-basic.sh (Terminal 2)**  
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-proxy-test-basic.json <http://localhost:9130/_/proxy/modules> HTTP/1.1 201 Created  
 Content-Type: application/json  
 Location: /\_/proxy/modules/test-basic-1.0.0  
 Content-Length: 527

The Okapi Gateway responds with an [HTTP 201 (“Created”)](https://httpstatusdogs.com/201-created) status code and returns the request body as the response body. There is also a Location:header that is the URL of the newly created module. That address can be used with other HTTP verbs (‘GET’ to read, ‘POST’ to update, ‘DELETE’ to remove) as one would expect for a RESTful resource. Run these commands to list the modules known to the gateway, delete test-module, and list the modules again to see for yourself:

**13-DeleteOkapi-proxy-test-basic.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET <http://localhost:9130/_/proxy/modules>$ curl -i -w '\n' -X DELETE <http://localhost:9130/_/proxy/modules/test-basic-1.0.0>$ curl -i -w '\n' -X GET <http://localhost:9130/_/proxy/modules>$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-proxy-test-basic.json <http://localhost:9130/_/proxy/modules>$ curl -i -w '\n' -X GET http://localhost:9130/\_/proxy/modules

### Deploying okapi-test-module

Although we have registered the module with the Okapi Gateway, we have not yet instantiated the module so that there is something available to respond to requests. The module must also be deployed on one node (or more, in the case of clusters). First, let’s query the discovery service for a list of nodes it knows about in the cluster:

**14-DiscoverAllNodes.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET <http://localhost:9130/_/discovery/nodes> HTTP/1.1 200 OK  
 Content-Type: application/json  
 Content-Length: 67  
 **[** **{**  
 "nodeId" : "localhost",  
 "url" : <http://localhost:9130> **}** **]**

The response body is a JSON document that, for our Okapi Gateway instance running with the ‘dev’ option, show just one node (nodeId of localhost).

To deploy the module, we create a “Deployment Descriptor” JSON document:

$ cat okapi-deploy-test-basic.json

Then post this document to the /\_/discovery core service:

**15-DeployOkapi-deploy-test-basic.sh (Terminal 2)**  
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-deploy-test-basic.json <http://localhost:9130/_/discovery/modules> HTTP/1.1 201 Created  
 Content-Type: application/json  
 Location: /\_/discovery/modules/test-basic-1.0.0/localhost-9131  
 Content-Length: 237

The Okapi Gateway server returns an [HTTP 201 (“Created”)](https://http.cat/201) status code responds with a JSON document that includes additional information:

* an instance identifier (‘instId’) for this particular deployment of the module
* a URL where the instance will receive requests
* a copy of the launch descriptor line from the Module Descriptor

In addition, the gateway returns a Location: header that is the URL of this deployed instance. As with module descriptors, that URL can be used to retrieve information about the deployed instance (HTTP GET) and stop the deployed instance (HTTP DELETE). Be sure to also flip back to the terminal session running the Okapi Gateway to see the server-side debug messages.

One important note: in our developer instance, we can access the okapi-test-module directly at the URL http://localhost:9131:

**16-CallTestbOkapi.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET <http://localhost:9131/testb>  
 HTTP/1.1 200 OK  
 Content-Type: text/plain  
 Content-Length: 8  
 It works

In a production system, there would be a firewall that prevents direct access to the Okapi Module interfaces. Requests for Okapi Module services must go through the Okapi Gateway to ensure that access control and tenant restrictions are honored.

## Creating a tenant

As noted above, all requests should be made through the Okapi Gateway. Let’s try that now:

$ curl -i -w '\n' -X GET <http://localhost:9130/testb>  
 HTTP/1.1 403 Forbidden  
 Content-Type: text/plain  
 Content-Length: 14  
 Missing Tenant

Okapi is inherently a multi-tenant system, so each request to an Okapi Module must be performed on behalf of a tenant. Let’s set up our first tenant:

**17-SetupTenantOkapi-tenant.sh (Terminal 2)**  
$ cat okapi-tenant.json   
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-tenant.json <http://localhost:9130/_/proxy/tenants> HTTP/1.1 201 Created  
 Content-Type: application/json  
 Location: /\_/proxy/tenants/testlib  
 Content-Length: 91

### Enable test-module for the testlib tenant

Now that the tenant is created, we need to enable test-module for that tenant:

**18-EnableTestModuleForTenant.sh (Terminal 2)**  
$ cat okapi-enable-basic.json   
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-enable-basic.json <http://localhost:9130/_/proxy/tenants/testlib/modules> HTTP/1.1 201 Created  
 Content-Type: application/json  
 Location: /\_/proxy/tenants/testlib/modules/test-basic-1.0.0  
 Content-Length: 31

Switch back to the Okapi Gateway terminal to see the debug output for this last request.

21:32:59 DEBUG ProxyContext 176769/\_ enableModule: Tenant init request to test-basic-1.0.0 succeeded  
21:32:59 INFO ProxyContext 176769/\_ RES 201 135657us okapi

In this string of messages you can see the Okapi Gateway finding the tenant and, as a “loopback” OkapiClient, post a request to the reserved /\_/tenant interface of okapi-test-module on http://localhost:9191 (http://localhost:9191/\_/tenant). This allows the Okapi Module to perform an initialization routine when it is added to a new tenant. Note the log line that says an X-Okapi-Tenant: testlib header is being added to the request being made by OkapiClient. We, too, must also add an X-Okapi-Tenantheader when sending a request to an Okapi Module:

**19-CallTestbAgainOkapi.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET -H 'X-Okapi-Tenant: testlib' http://localhost:9130/testb  
 HTTP/1.1 200 OK  
 Content-Type: text/plain  
 X-Okapi-Trace: GET - Okapi test module http://localhost:9131/testb : 200 12629us  
 Transfer-Encoding: chunked  
 It works

Also note that the core interfaces behave as we expect them to by returning JSON lists. Try these commands:

**20-ListAllModulesForTestbOkapi.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET <http://localhost:9130/_/proxy/tenants>  
$ curl -i -w '\n' -X GET http://localhost:9130/\_/proxy/tenants/testlib/modules

# 03: Enable Okapi Authentication

All real-world uses of Okapi will need to provide authentication and authorization services. The Okapi Gateway itself does not handle these tasks; rather, it delegates them to one or more Okapi Modules that operates at an early phase of module requests orchestrated by the Okapi Gateway. In this lesson we will first interact directly with a sample authentication Okapi Module. Later in the lesson we will register and deploy the sample authentication module on the Okapi Gateway. Authorization will be covered in a later lesson.

## Interact Directly with Okapi-test-auth-module

Okapi-test-auth-module is a simple module that illustrates authentication in Okapi, and it comes with okapi-core.

In a terminal window, start the okapi-test-auth-module.

When using the VirtualBox method, you will need to open a terminal window on your host computer, change the working directory to the location of the Vagrantfile, and use the vagrant ssh command to connect from the host computer to the guest.

**21-RunOkapiTestAuthModule.sh (Terminal 1 – CTRL+C)**  
$ cd $FOLIO\_ROOT/okapi  
$ java -jar okapi-test-auth-module/target/okapi-test-auth-module-fat.jar  
 02:15:42 INFO MainVerticle Starting auth 26062@contrib-jessie on port 9020  
 02:15:42 INFO ertxIsolatedDeployer Succeeded **in** deploying verticle

The /login path of okapi-test-auth-module takes a simple JSON document with a ‘username’ and a ‘password’ value. In this test authentication module, if the password is the same as the username with the string ‘-password’ appended then the authentication is successful and a token is returned. This can be demonstrated by using this curl command in a second terminal window to send a sample JSON document to the okapi-test-auth-module.

**22-LoginToTestAuth.sh (Terminal 2)**  
$ curl -i -w '\n' -X POST -H 'X-Okapi-Tenant: blah' \  
 -d '{"username": "seb", "password": "seb-password"}' \  
 <http://localhost:9020/authn/login> HTTP/1.1 200 OK  
 Content-Type: application/json  
 X-Okapi-Token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig  
 Content-Length: 47  
 **{**"username": "seb", "password": "seb-password"**}**

Now try to send a JSON document in which the password does not match the expected value. Note that the response returns an [HTTP 401 “Unauthorized”](https://http.cat/401) status.

Another path supplied by okapi-test-auth-module is /check; it checks an X-Okapi-Token to see if it is valid.

**23-CheckTokenTestAuth.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET -H 'X-Okapi-Tenant: blah' \  
 -H 'X-Okapi-Token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig' \  
 http://localhost:9020/check  
 HTTP/1.1 202 Accepted  
 X-Okapi-Token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig  
 X-Okapi-Module-Tokens: **{}**  
 Content-Type: text/plain  
 Transfer-Encoding: chunked

Note that the response returns an [HTTP 202 “Accepted”](https://httpstatusdogs.com/202-accepted) status. Try sending a request without the required X-Okapi-Tenant header. All requests are expected to send the X-Okapi-Tenant header.

Return to the terminal window with okapi-test-auth-module running and press Control-C to exit it.

## Register and Deploy the Okapi-test-auth-modulein the Okapi Gateway

The Okapi-test-auth-module is registered and deployed just like any other Okapi Module: using a ModuleDescriptor and a DeploymentDescriptor. In the terminal window, start the Okapi Gateway with debugging turned on:

**24-RunInitOkapi.sh (Terminal 1 – Ctrl+C Select n) (Terminal 2 select y)**  
  
$ cd $FOLIO\_ROOT/okapi  
$ java -Dloglevel**=**DEBUG -jar okapi-core/target/okapi-core-fat.jar dev  
 12:08:11 INFO MainVerticle git: git@github.com:folio-org/okapi 225c9c1e03c29459da430f93110abb30378e1394  
 12:08:11 INFO MainVerticle clusterManager not **in** use  
 12:08:11 INFO MainVerticle Proxy using inmemory storage  
 12:08:12 WARN Storage Storage.resetDatabases: NORMAL  
 12:08:12 INFO TenantWebService All tenants deployed  
 12:08:12 INFO MainVerticle API Gateway started PID 64161@Walkabout.lan. Listening on port 9130

Open up a second terminal window to send JSON documents to the Okapi Gateway using curl (noting that if you are VagrantBox method you will need to open a new terminal on your host and use the vagrant ssh command). The examples in this tutorial are using the in-memory storage, so you will need to resend the Okapi-test-module ModuleDescriptor and DeploymentDescriptor as well as redefine the testlib tenant and enable the Okapi-test-module. If you completed the previous lesson, these JSON documents are in the $FOLIO\_ROOT directory.

$ cd $FOLIO\_ROOT  
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-proxy-test-basic.json <http://localhost:9130/_/proxy/modules>$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-deploy-test-basic.json <http://localhost:9130/_/discovery/modules>$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-tenant.json <http://localhost:9130/_/proxy/tenants>$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-enable-basic.json  
http://localhost:9130/\_/proxy/tenants/testlib/modules

### Okapi Auth Module ModuleDescriptor

This example of a ModuleDescriptor is similar to the one used for the Okapi-test-module.

**25-RegisterOkapi-proxy-test-module-auth.1.sh (Terminal 2)**  
$ cat okapi-proxy-test-module-auth.json   
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-proxy-test-module-auth.json http://localhost:9130/\_/proxy/modules

The significant difference is a new dictionary called filters. The filtersdictionary specifies paths and phases for which the Okapi Gateway will call this module. At the moment, there is only one phase (auth) and the methodsand pathPattern specify that this module will be called for all Okapi Gateway requests. Another difference is that the launchDescriptor is omitted; as shown below, the launchDescriptor can also be defined in the DeploymentDescriptor. (This can be useful in cases where there are command-line options specific to a particular deployment.)

### Okapi Auth Module DeploymentDescriptor

This example of a DeploymentDescriptor is similar to the one used for the Okapi-test-module. Note that the launchDescriptor dictionary is defined here.

**26-DeployOkapi-deploy-test-module-auth.sh (Terminal 2)**  
$ cat okapi-deploy-test-module-auth.json   
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-deploy-test-module-auth.json http://localhost:9130/\_/discovery/modules

### Enable Okapi Auth Module on tenant

**27-EnableOkapi-enable-auth.sh (Terminal 2)**  
$ cat okapi-enable-auth.json   
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-enable-auth.json http://localhost:9130/\_/proxy/tenants/testlib/modules

### Demonstrating the Impact of the Okapi Auth Module

If we now send the same test request from the end of the previous lesson to the Okapi-test-module, we see a different response from the Okapi Gateway.

**28-CallTestbWithTokenOkapi.sh (Terminal 2)**  
$ curl -i -w '\n' -X GET -H 'X-Okapi-Tenant: testlib' <http://localhost:9130/testb> HTTP/1.1 401 Unauthorized  
 Content-Type: text/plain  
 X-Okapi-Trace: GET Okapi test auth module http://localhost:9132/testb : 401 95343us  
 Transfer-Encoding: chunked  
 Auth.check called without X-Okapi-Token

With the Okapi-test-auth-module filter now configured to intercept all requests to the Okapi Gateway, the Gateway responds with an [HTTP 401 (“Unauthorized”)](https://http.cat/401) and in the response body tells us that the X-Okapi-Token is missing. This is the same X-Okapi-Token described in the first section of this lesson. If we send a request header with a valid X-Okapi-Token, the request is successful.

$ curl -i -w '\n' -X GET -H 'X-Okapi-Tenant: testlib' \  
 -H 'X-Okapi-Token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig' \  
 <http://localhost:9130/testb> HTTP/1.1 200 OK  
 X-Okapi-Trace: GET Okapi test auth module http://localhost:9132/testb : 202 188522us  
 Content-Type: text/plain  
 X-Okapi-Trace: GET Okapi test auth module http://localhost:9132/testb : 202 188522us  
 X-Okapi-Trace: GET Okapi test module http://localhost:9131/testb : 200 10933us  
 Transfer-Encoding: chunked  
 It works

# 04: Deploy test Stripes module

In this lesson we set aside the Okapi server and focus on the web-based user interface, Stripes. The Okapi Gateway is not required for this lesson.

## Inform the Yarn package manager of the FOLIO UI registry location

(Note: the output of commands is artificially indented from the command line to call out the command lines.)

**29-InformYarn.sh (Executed)**  
$ yarn config set @folio:registry https://repository.folio.org/repository/npm-folio/  
 yarn config v0.20.3  
 success Set "@folio:registry" to "https://repository.folio.org/repository/npm-folio/".  
 ✨ Done in 0.04s.

## Set up a Stripes Development UI Server

Create an empty directory to hold the Stripes UI Server configuration (called stripes-tutorial-platform).

$ mkdir $FOLIO\_ROOT/stripes-tutorial-platform  
$ cd $FOLIO\_ROOT/stripes-tutorial-platform

In that directory, put two files: package.json and stripes.config.js.

### Contents of package.json

package.json is a [Node Package Manager (NPM) configuration file](https://docs.npmjs.com/files/package.json). It is a JSON file that contains two dictionaries: scripts and dependencies. The scriptsdictionary specifies name-value pairs of commands that are used by Yarn to build and run the platform. The dependencies dictionary lists packages (and specific versions) that make up the Stripes client bundles.

At this stage of the Curriculum we are setting up a stand-alone Stripes UI Server instance that does not communicate with an Okapi back-end. The package.json below builds Stripes with a ‘trivial’ client bundle.

{} **NOTE: This section is updated for the VALATechCamp17**- Copy the content of stripes-tutorial-platform\standalone\_package and paste it into stripes-tutorial-platform

### Contents of stripes.config.js

stripes.config.js contains the configuration details for the Stripes UI Server. It is referenced in scripts dictionary of package.json. It is a JSON file with three required dictionaries: okapi, config and modules. The okapidictionary specifies the details for connecting to the Okapi Gateway; it is not used in this lesson. The config dictionary contains two key-value pairs to bypass authentication and authorization checks. The modules dictionary contains another dictionary of Stripes package and their configuration. The key in this dictionary is the name of the package to load from the FOLIO UI registry. The value in this dictionary are parameters that can override the default settings of the Stripes package.

### Build Stripes with the ‘trivial’ client bundle

Download/update Stripes along with its dependencies and modules, and link them together using the yarn install command:

**30-InstallYarn.sh (Executed)**  
$ yarn install  
 yarn install v0.20.3  
 info No lockfile found.  
 warning No license field  
 **[**1/4] 🔍 Resolving packages...  
 **[**2/4] 🚚 Fetching packages...  
 **[**3/4] 🔗 Linking dependencies...  
 **[**4/4] 📃 Building fresh packages...  
 success Saved lockfile.  
 ✨ Done **in** 40.40s.

If you are seeing the error “EPROTO: protocol error, symlink” when running Vagrant on Windows, the likely cause is that Vagrant was not launched with administrator privileges. Be sure to use Run As Administrator… when opening the Command Prompt itself (cmd.exe). See issue [STRIPES-344](https://issues.folio.org/browse/STRIPES-344) for details.

After the Stripes UI Server is built, run it using the yarn start command:

**31-StartYarn.sh (Terminal 1 Ctrl+C)**  
$ yarn start  
 yarn start v0.20.3  
 $ stripes dev stripes.config.js  
 Listening at <http://localhost:3000> webpack built 554cedd72fbedc2f7499 **in** 7890ms

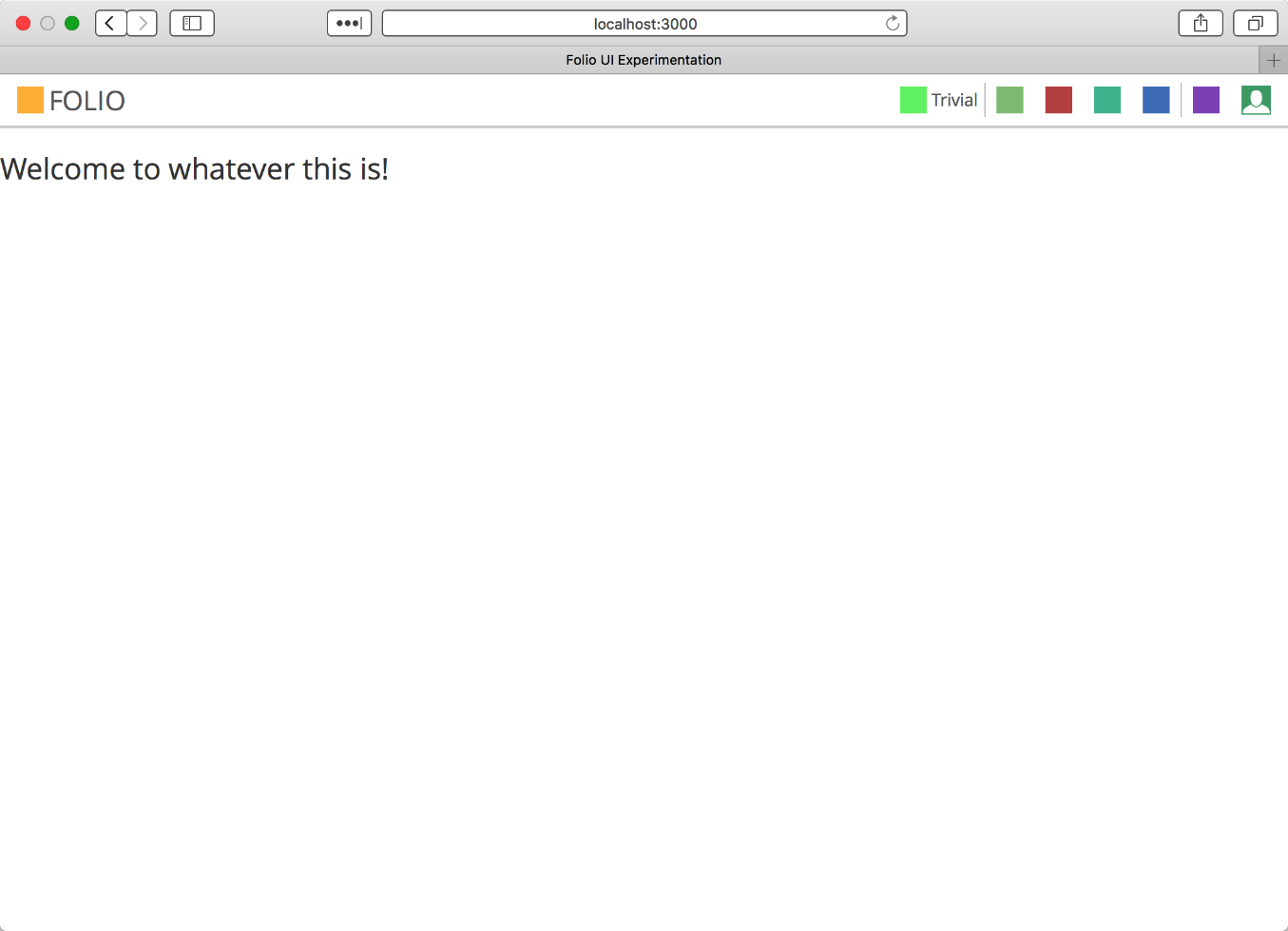
If you are using the VirtualBox guest machine, set the environment variable STRIPES\_HOST before running yarn start to allow the Stripes development server to listen on all interfaces:

$ STRIPES\_HOST=0.0.0.0 yarn start

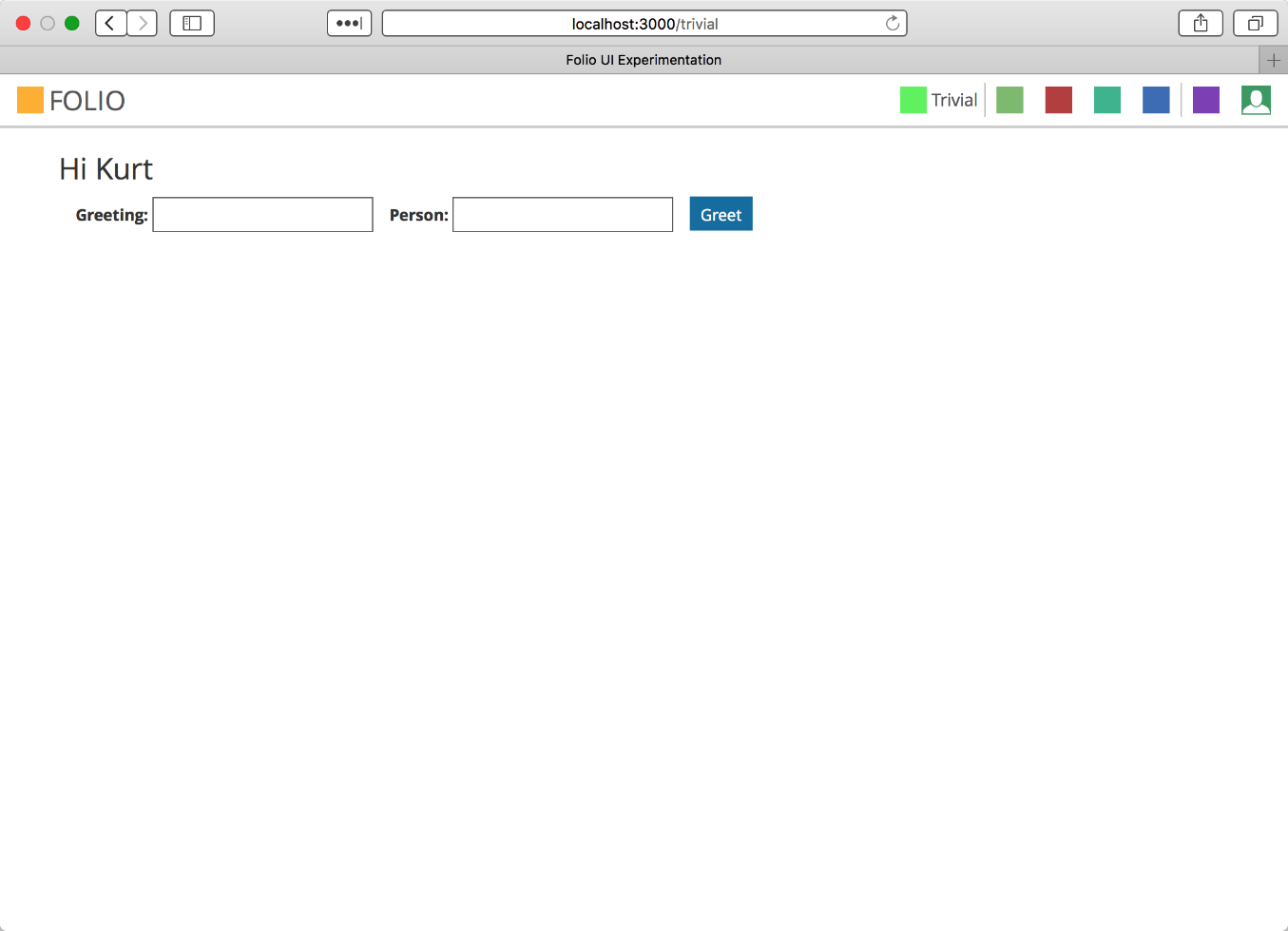
The Stripes UI Server is now running at http://localhost:3000. The server will respond after the webpack built... message is displayed.

## Interacting with the Stripes UI Server

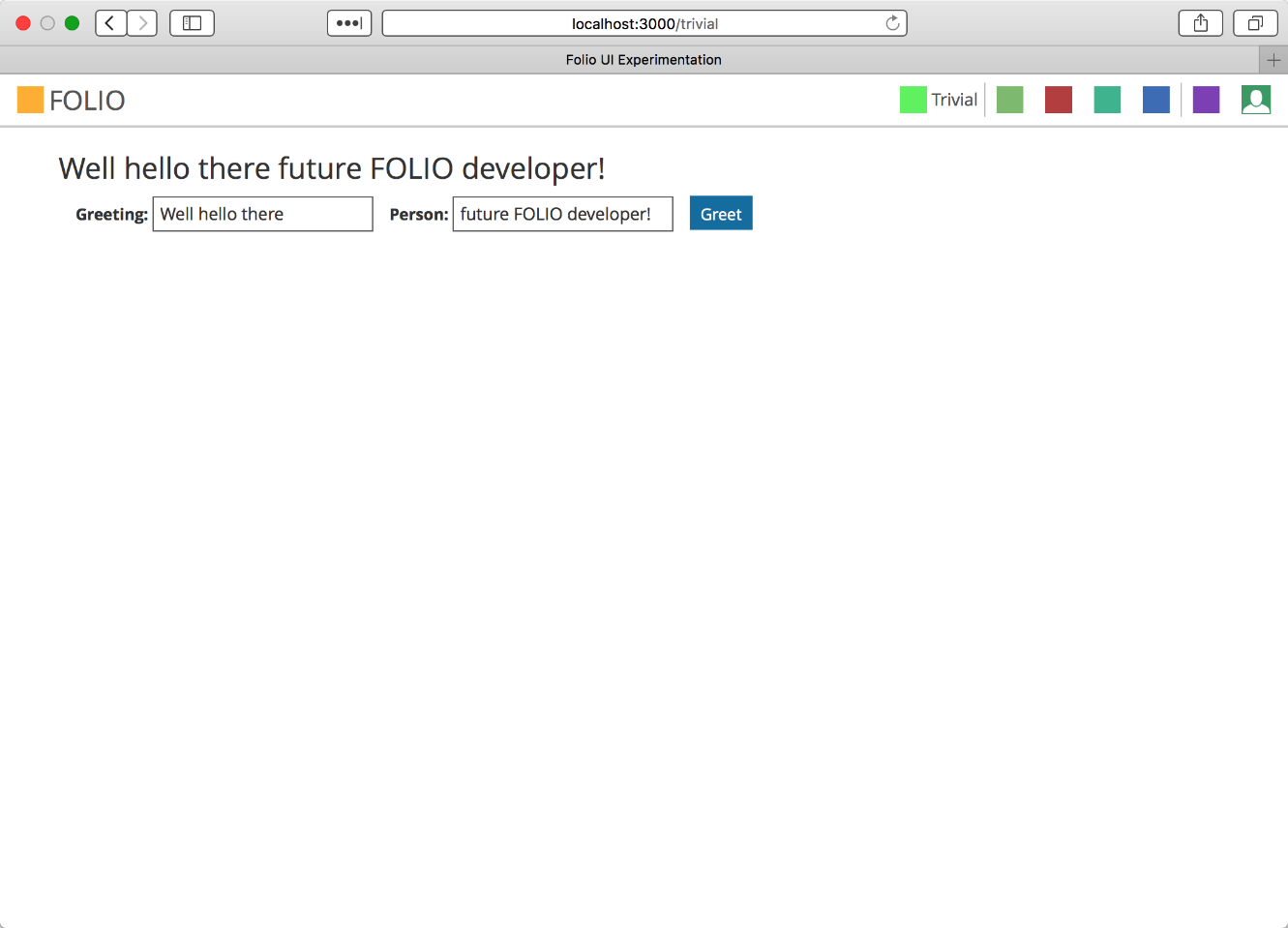
The Stripes UI Server homepage at http://localhost:3000 looks like the figure below.



There is one app in the Stripes UI Server – the “Trivial” app with the green icon. Click on it to get a form:

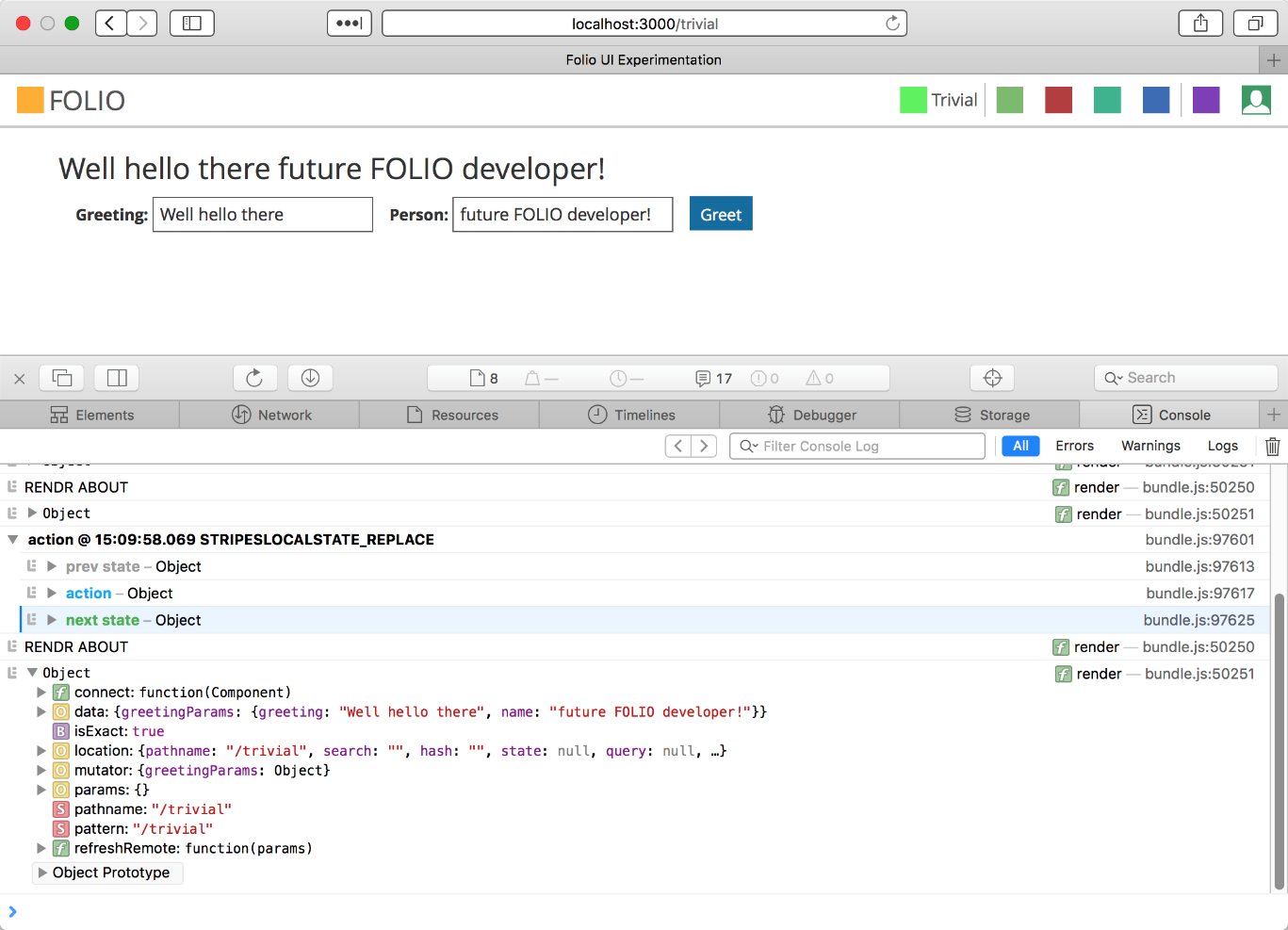


Type in a greeting and name of your choice and submit the form to see the reply.



This is an example of the Stripes server component communicating with a Stripes browser component. We have not set up the Okapi part of the FOLIO system, so this interaction is strictly within the Stripes UI Server itself.

The source for the Trivial module is in the stripes-core git repository (https://github.com/folio-org/stripes-core/tree/master/examples/trivial), with the bulk of the work in the [About.js](https://github.com/folio-org/stripes-core/blob/master/examples/trivial/About.js) file. More details about the state of the object within the module can be seen by viewing the debugging output in the browser’s JavaScript console.



# 05: Set up the Okapi Users app

In lesson one, we deployed Stripes and demonstrated communication between the browser and the Stripes components. In lessons two and three, we deployed the Okapi Gateway as well as a test Okapi Module and examined the communication between them. In lesson four we are connecting Stripes to the Okapi Gateway and adding the Users app. There are two components to the Users app: the Stripes UI component and the Okapi Module component. We will start first with the Stripes UI component.

## Add the Users app UI component to the Stripes UI Server

Remember in $FOLIO\_ROOT/stripes-tutorial-platform we have two configuration files: package.json and stripes.config.js. Each will need changes to add the Users app.

### Modify package.json

The package.json file needs a new entry in the dependency section. The revised file should look like:

{} **NOTE: This section is updated for the VALATechCamp17**- Copy the content of stripes-tutorial-platform\user\_package and paste it into stripes-tutorial-platform

### Modify stripes.config.js

The stripes.config.js file needs not only a line for adding the Users UI component; it also needs connection information for the Okapi Gateway.

The okapi line in stripes.config.js gives the Stripes UI Server the URL of the Okapi gateway and the name of the tenant being used by this UI Server instance.

### Rebuild Stripes Server

With the Users UI components added to the UI Server configuration, use the yarn install command to download and configure the necessary modules.

When using the VirtualBox method, you will need to open a terminal window on your host computer, change the working directory to the location of the Vagrantfile, and use the vagrant ssh command to connect from the host computer to the guest.

When starting Stripes from the command line, be sure to set the STRIPES\_HOST environment variable: STRIPES\_HOST=0.0.0.0 yarn start

**30-InstallYarn.sh (Terminal 3)**  
$ cd $FOLIO\_ROOT/stripes-tutorial-platform  
$ yarn install  
 yarn install v0.20.3  
 warning No license field  
 **[**1/4] 🔍 Resolving packages...  
 **[**2/4] 🚚 Fetching packages...  
 **[**3/4] 🔗 Linking dependencies...  
 warning "eslint-import-resolver-webpack@0.8.1" has unmet peer dependency "eslint-plugin-import@>=1.4.0".  
 **[**4/4] 📃 Building fresh packages...  
 success Saved lockfile.  
 ✨ Done **in** 12.12s.  
$ yarn start  
 yarn start v0.20.3  
 $ stripes dev stripes.config.js  
 Listening at <http://localhost:3000> webpack built 97b8243748d40fd3a9c1 **in** 14004ms

The Stripes portion of the Users app is now running at<http://localhost:3000/users>.

## Add the Users app Okapi Module to the Okapi Gateway

### Fetch and build the Users app Okapi Module

**32-GitCloneModUsers.sh (Terminal 1 Ctrl+C)**  
$ cd $FOLIO\_ROOT  
$ git clone --recursive <https://github.com/folio-org/mod-users.git> Cloning into 'mod-users'...  
  
**33-MvnInstallModUsers.sh (Executed)**   
$ cd mod-users  
$ mvn install  
 **[**...]  
 **[**INFO] ------------------------------------------------------------------------  
 **[**INFO] BUILD SUCCESS  
 **[**INFO] ------------------------------------------------------------------------  
 **[**INFO] Total time: 01:21 min  
 **[**INFO] Finished at: 2017-02-24T16:52:58-05:00  
 **[**INFO] Final Memory: 74M/532M  
 **[**INFO] ------------------------------------------------------------------------

### Register and Deploy the Users app Okapi Module

The Git repository for the Users app Okapi Module has a [Module Descriptor](https://github.com/folio-org/mod-users/blob/master/ModuleDescriptor.json)and a [Deployment Descriptor](https://github.com/folio-org/mod-users/blob/master/DeploymentDescriptor.json) that can be used to register and deploy the Users app Okapi Module.

**34-RunInitOkapi.sh (Terminal 1 – Ctrl+C Select n) (Terminal 2 select y)**

**35-RegisterModUser.sh (Terminal 2)**  
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @ModuleDescriptor.json <http://localhost:9130/_/proxy/modules> HTTP/1.1 100 Continue  
 HTTP/1.1 201 Created  
 Content-Type: application/json  
 Location: /\_/proxy/modules/users-module  
 Content-Length: 4118

You will also need to deploy the module with a Deployment Descriptor:

**36-DeployModUser.sh (Terminal 2)**  
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @DeploymentDescriptor.json <http://localhost:9130/_/discovery/modules> HTTP/1.1 201 Created  
 Content-Type: application/json  
 Location: /\_/discovery/modules/users-module/localhost-9131  
 Content-Length: 245

(Note: Your port number in the instId and the url may vary depending on whether there are other Okapi Modules deployed on the Okapi Gateway.) Finally, you’ll need to enable the Okapi Users app module for the test tenant:

**37-EnableOkapi-enable-users.sh (Terminal 2)**  
$ cat okapi-enable-users.json   
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -d @okapi-enable-users.json  
<http://localhost:9130/_/proxy/tenants/testlib/modules>HTTP/1.1 201 Created  
Content-Type: application/json  
Location: /\_/proxy/tenants/testlib/modules/users-module  
Content-Length: 27

The FOLIO Users app is now available at <http://localhost:3000/users>. You’ll see the message No results found for "". Please check your spelling and filters because no users have been added.

## Add users to FOLIO

For testing purposes, the FOLIO development team has JSON documents representing factitious users that can be used to populate the dev FOLIO environment.

**39-StartYarn.sh** *(check terminal 3 to confirm execution is complete. Run in terminal 2)*  
**38-AddUser.sh (Optional)**  
$ cd $FOLIO\_ROOT  
$ cat User000.json   
$ cat User001.json   
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -H 'X-Okapi-Token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig' \  
 -H 'X-Okapi-Tenant: testlib' -d @User000.json <http://localhost:9130/users> HTTP/1.1 201 Created  
 X-Okapi-Trace: POST Okapi test auth module http://localhost:9132/users : 202  
 Content-Type: application/json  
 Location: 48b6982e-3504-4bfe-8fa6-f16d686d9d5a  
 user-agent: curl/7.38.0  
 host: localhost:9130  
 accept: **\***/**\***  
 x-okapi-tenant: testlib  
 x-okapi-request-id: 408458/users  
 x-okapi-url: <http://localhost:9130> x-okapi-permissions-required: users.item.post  
 x-okapi-module-permissions: **{}**  
 x-okapi-token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig  
 X-Okapi-Trace: POST Okapi test auth module http://localhost:9132/users : 202  
 X-Okapi-Trace: POST users http://localhost:9133/users : 201 51570us  
 Transfer-Encoding: chunked  
$ curl -i -w '\n' -X POST -H 'Content-type: application/json' \  
 -H 'X-Okapi-Token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig' \  
 -H 'X-Okapi-Tenant: testlib' -d @User001.json <http://localhost:9130/users> HTTP/1.1 201 Created  
 X-Okapi-Trace: POST Okapi test auth module http://localhost:9132/users : 202  
 Content-Type: application/json  
 Location: e30d2586-f90d-4dee-8e05-afd72e0d65aa  
 user-agent: curl/7.38.0  
 host: localhost:9130  
 accept: **\***/**\***  
 x-okapi-tenant: testlib  
 x-okapi-request-id: 587880/users  
 x-okapi-url: <http://localhost:9130> x-okapi-permissions-required: users.item.post  
 x-okapi-module-permissions: **{}**  
 x-okapi-token: dummyJwt.eyJzdWIiOiJzZWIiLCJ0ZW5hbnQiOm51bGx9.sig  
 X-Okapi-Trace: POST Okapi test auth module http://localhost:9132/users : 202  
 X-Okapi-Trace: POST users http://localhost:9133/users : 201 22312us  
 Transfer-Encoding: chunked

You can also use the New User button in the browser interface to add a user.