Cloud Developer Certification Preparation

Exercise 3.4:
Load testing cloud applications

Exercise 3.4: Prerequisites

Sign up for a 30-day free trial IBM Bluemix account if you don't already have one.

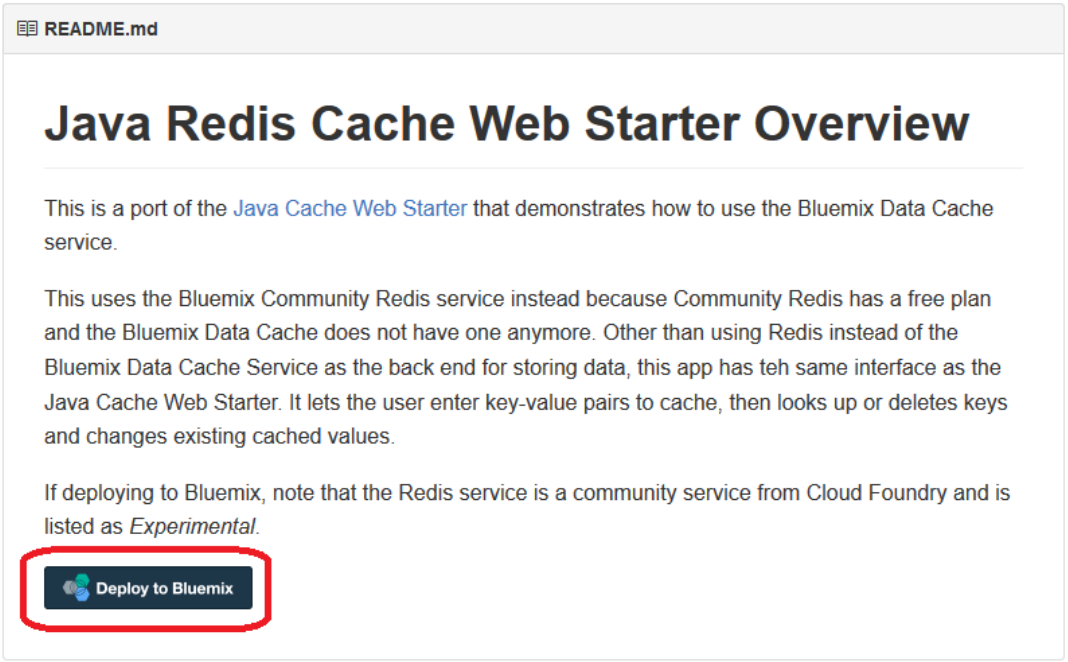
You also need the following software:

- A web browser supported by Bluemix:
    - Chrome: the latest version for your operating system
    - Firefox: the latest version for your operating system and ESR 31 or ESR 38
    - Internet Explorer: version 10 or 11
    - Safari: the latest version for the Mac

Exercise 3.4.1: Importing the application used for load testing from Github

Complete the following steps to import the application:

1. In your browser, go the URL https://github.com/ibmecod/java-rediscache and click **Deploy to Bluemix**.



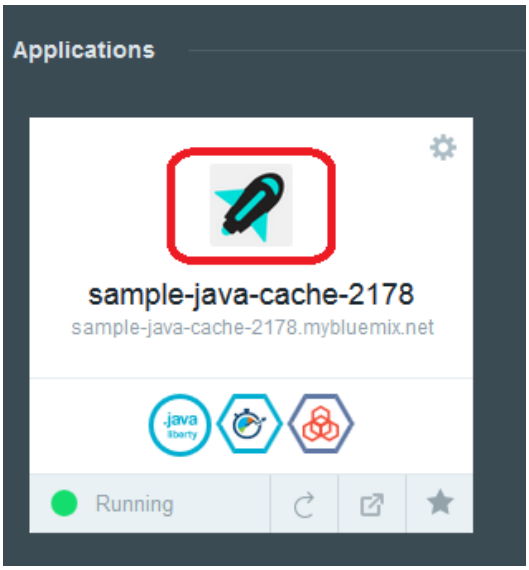
2. Click **LOGIN** if prompted and the click **DEPLOY** to deploy the app to your Bluemix account.

3. Wait for the message that says `SUCCESS! You've added an instance of this app to your organization in Bluemix` and then click the **DASHBOARD** link at the top of the page.

Exercise 3.4.2: Adding the Auto-Scaling service to your application

Complete the following steps to add the **Auto-Scaling** service:

1. In the Bluemix dashboard, click the icon for the app you just imported.



2. Click **ADD A SERVICE OR API.** This will show the catalog of services.

3. Scroll to the **DevOps** section and select the **Auto-Scaling** service.

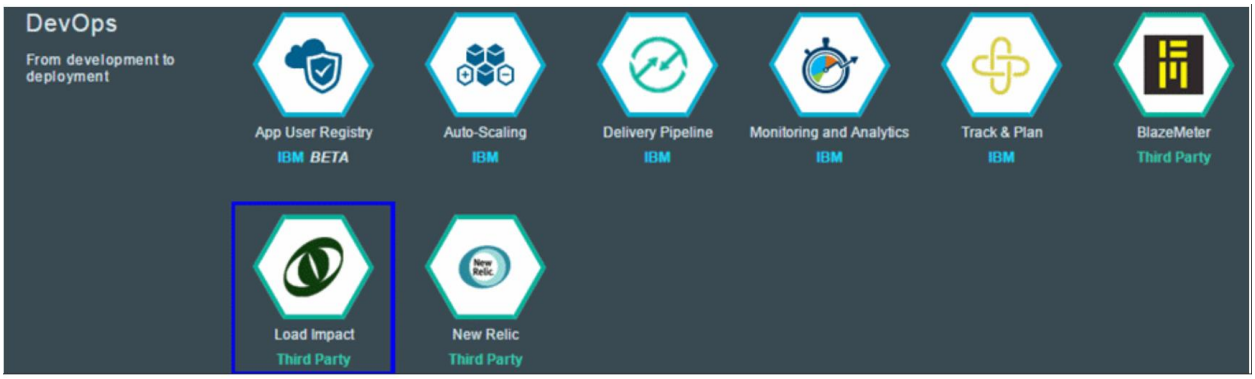
4. Accept the default values and click **CREATE**. Click **RESTAGE** when prompted.

5. Wait for the message that says `Your app is running` and then click **Back to Dashboard** at the top of the page.

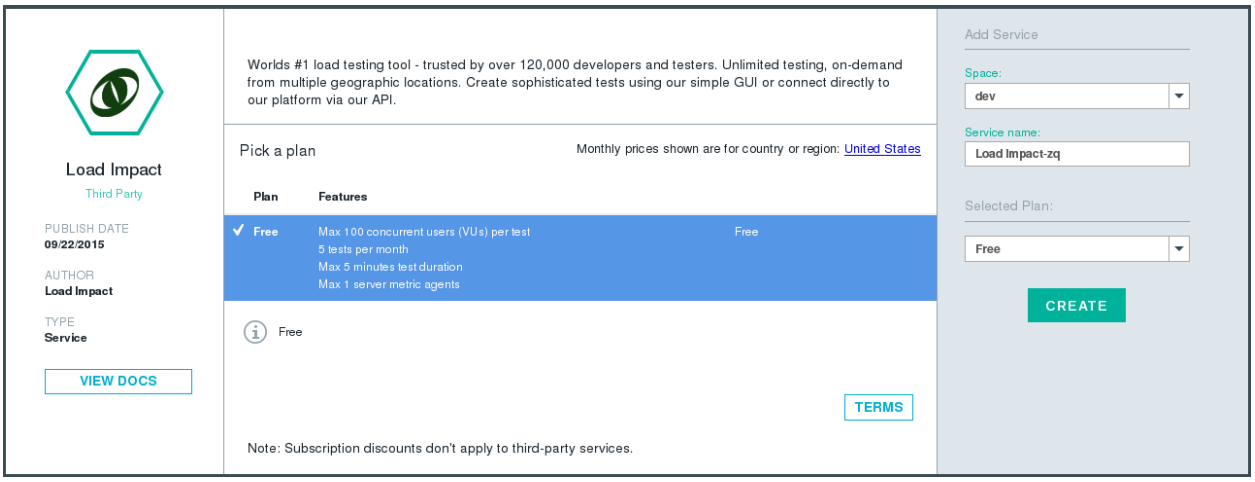
Exercise 3.4.3: Adding a Load Impact service to your application

Complete the following steps to add a Load Impact service:

1. In the Bluemix dashboard, click **CATALOG**. Navigate to the DevOps section and select the **Load Impact** service.



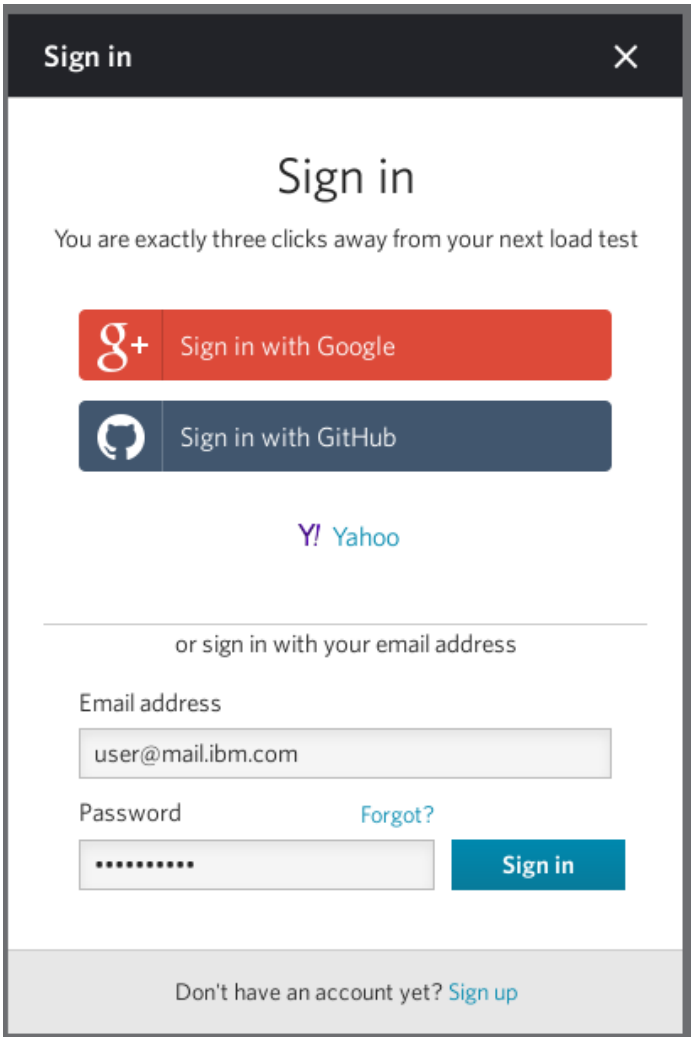
2. Click **CREATE** to create the service and add to your space. This service instance can be used across multiple applications. Unlike other services, it is not bound to a specific application. Accept the default values.



After the service is added to the dashboard, the service control panel is displayed with a link to **Open Load Impact Dashboard**.

3. Open the **Load Impact** Dashboard.

4. When prompted, create a new Load Impact account or sign in by using your current account.



After you sign in, the Load Impact web control page is displayed for you to configure the load test.

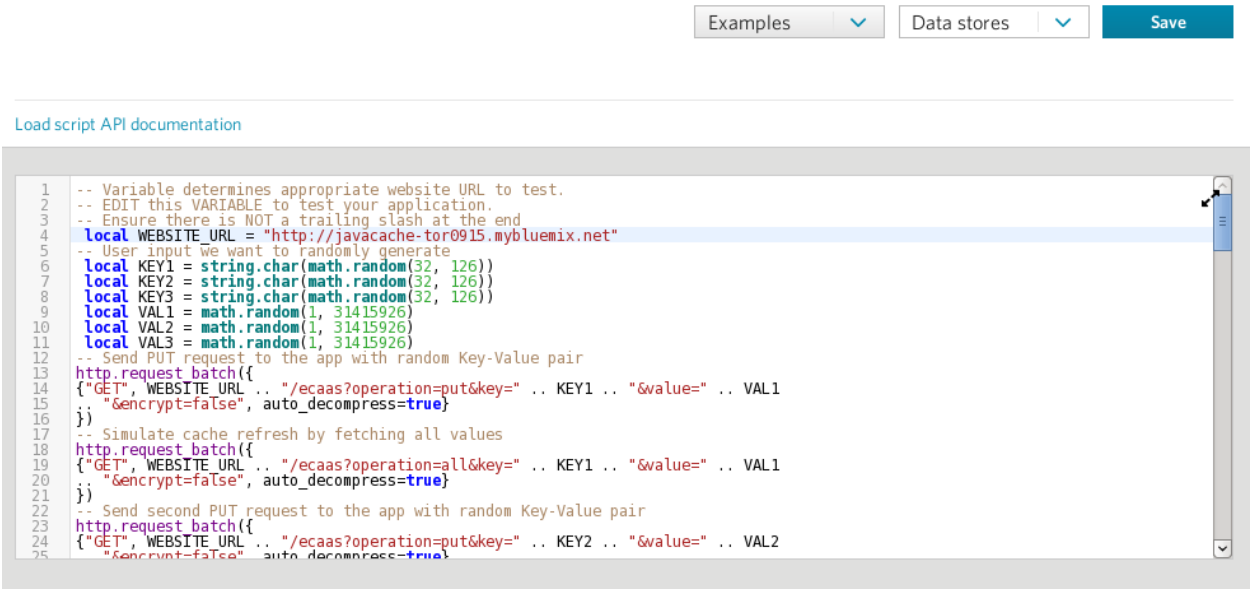
Exercise 3.4.4: Creating a user scenario

A load impact user scenario defines what actions a simulated user will take during the load tests of your application. The Load Impact service can create auto-generated tests from a URL, but to achieve more realistic tests for your application, creating your own tests is important. Start by defining a new user scenario with the following steps:

1. Click the **User Scenarios** icon on the left side. Then, click **New user scenario**.

2. Select the Scripting option. This exercise includes a sample script in Exercise 3.4.7.

3. Paste into the text box the script from Exercuse 3.4.7.



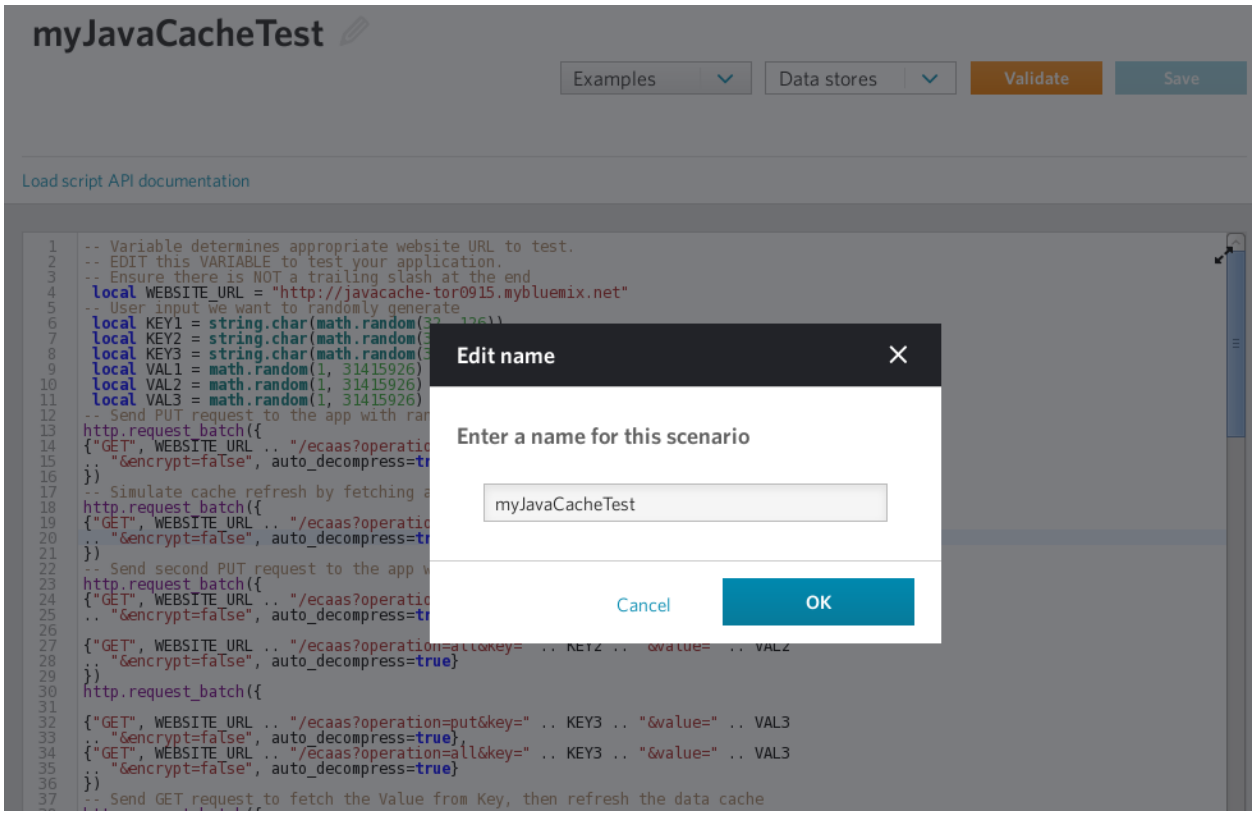
4. Change the target in the WEBSITE_URL variable to match your Bluemix application.

5. Click **Validate** to make sure that the script does not have errors. You might see some URL warnings, which are OK.

6. Click **Save**.

7. Optional: Change the name of the user scenario by clicking the Pencil icon and updating the name.

Exercise 3.4: Load testing



The user scenario provided in the exercise was created with a combination of output from the Load Impact Chrome recorder extension and some custom Lua scripting. For more information about generating your own tests for other applications, see the Load Impact documentation:

- Load Impact Quick Start Guide: http://support.loadimpact.com/knowledgebase/articles/302067-load-impact-quick-start-guide
- Simulating realistic load using Load Impact: http://support.loadimpact.com/knowledgebase/articles/265464-simulating-realistic-load-using-load-impact
- Lua Quick Start Guide: http://support.loadimpact.com/knowledgebase/articles/174637-lua-quick-start-guide

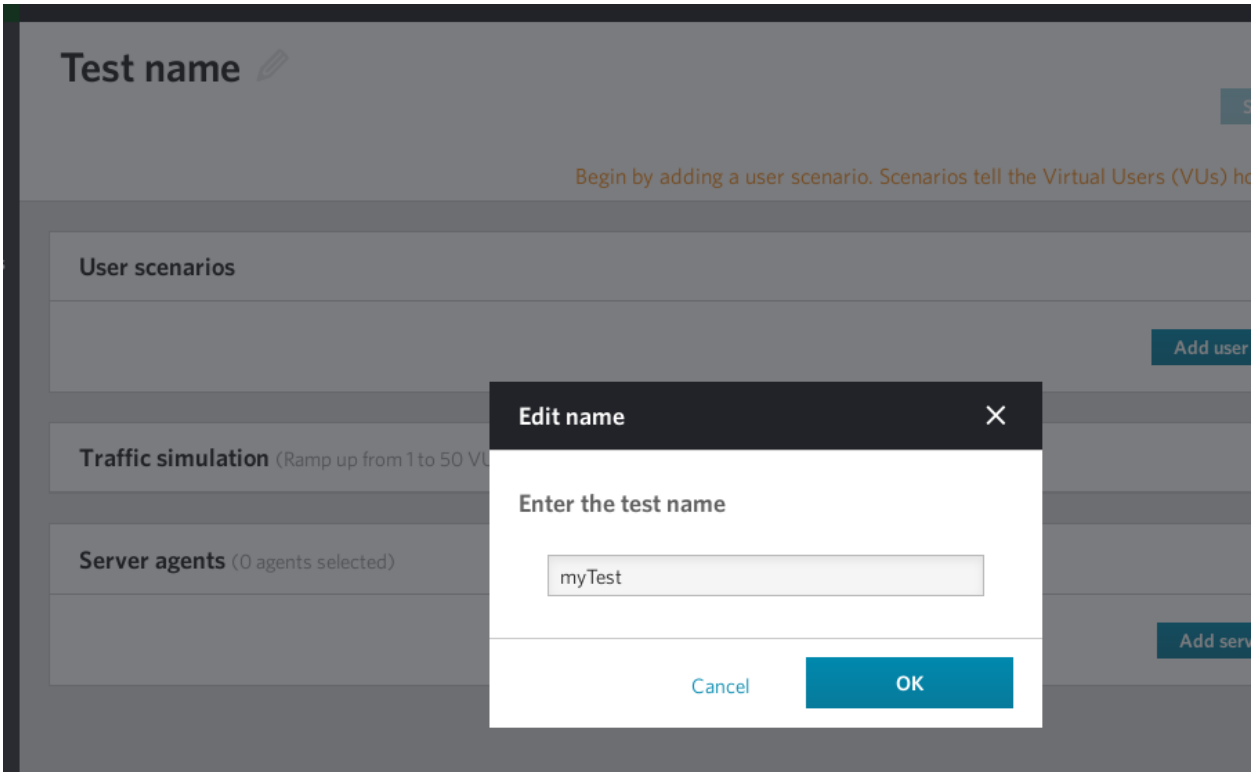
Exercise 3.4.5: Creating and running a test configuration

A test configuration in Load Impact orchestrates how simulated users will act and behave during a Load Impact test. Here you define testing factors such as how many users will be simulated, how long the test will run, the number of servers used, and which user scenarios will be run during the test.

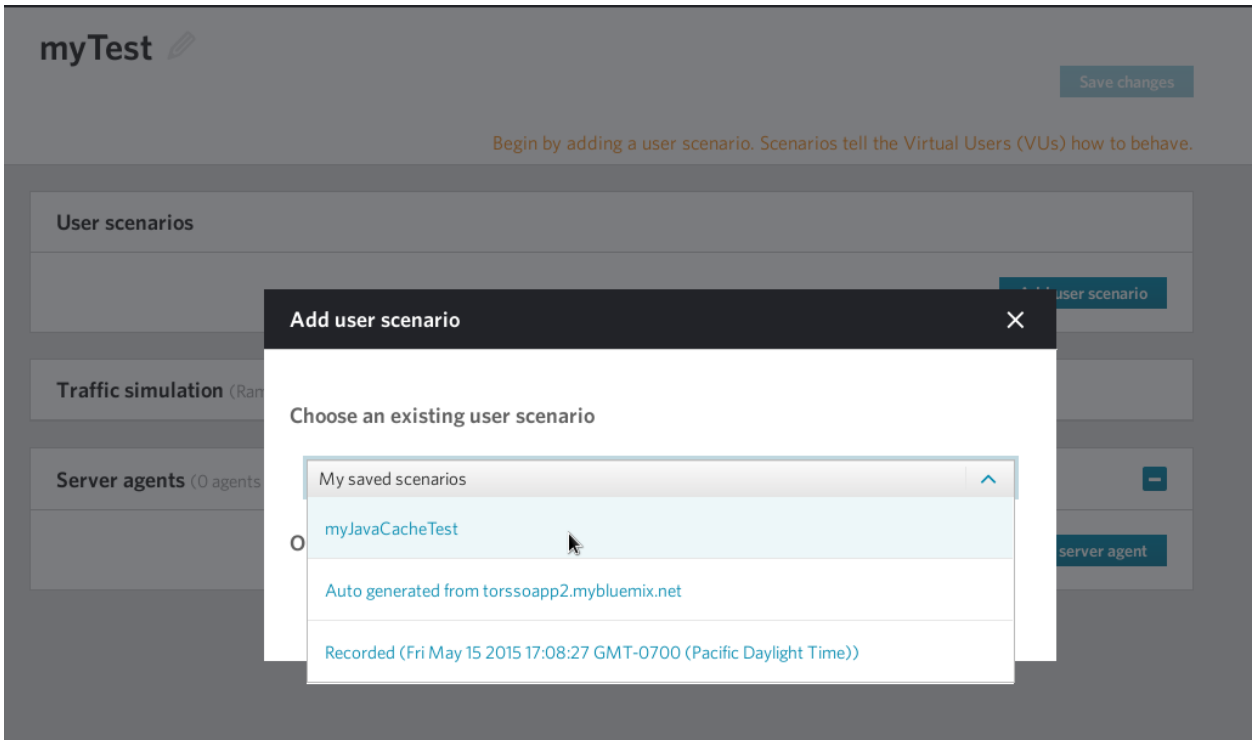
1. Click the **Tests** icon on the left side. Then, click **New Test**.

2. Click the upper-right **Skip to advanced mode** text.

3. Click the **Pencil** name next to the words **Test name** to to enter a new name.

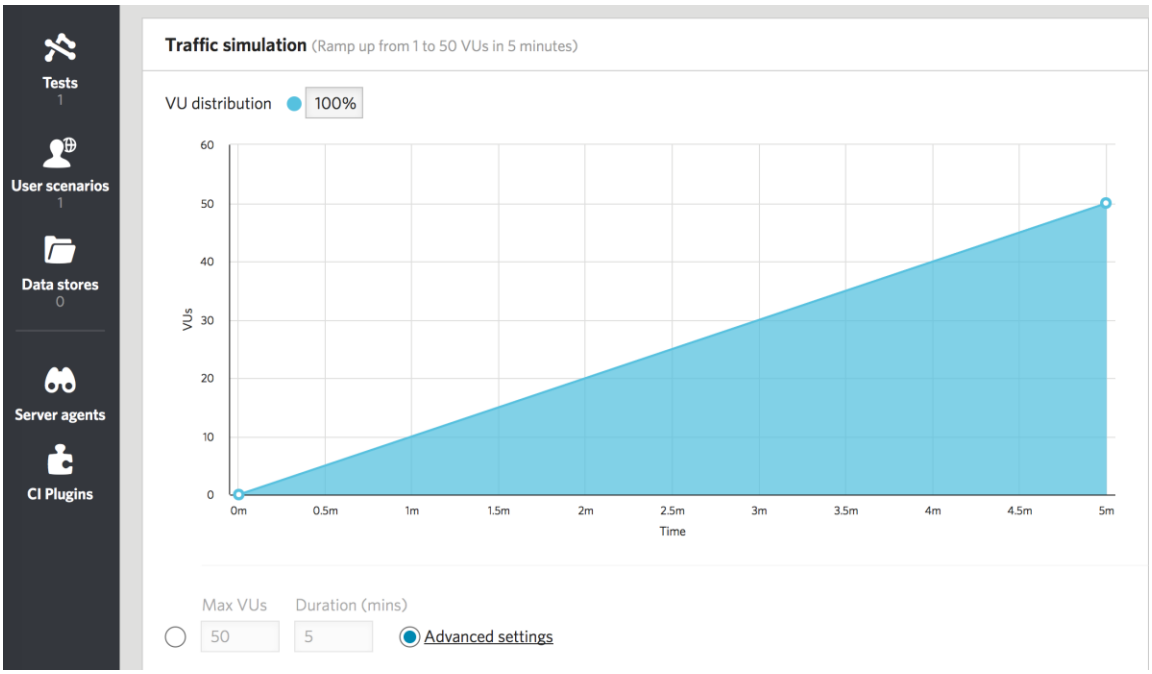


4. Click **Add user scenario** and select the scenario that you created in the previous section.



A window shows a proposed load test using the scenario. You can keep the Load zone at the default location.

5. Scroll down and note that in the proposed test, the user load will ramp up to 50 users over the span of 5 minutes.

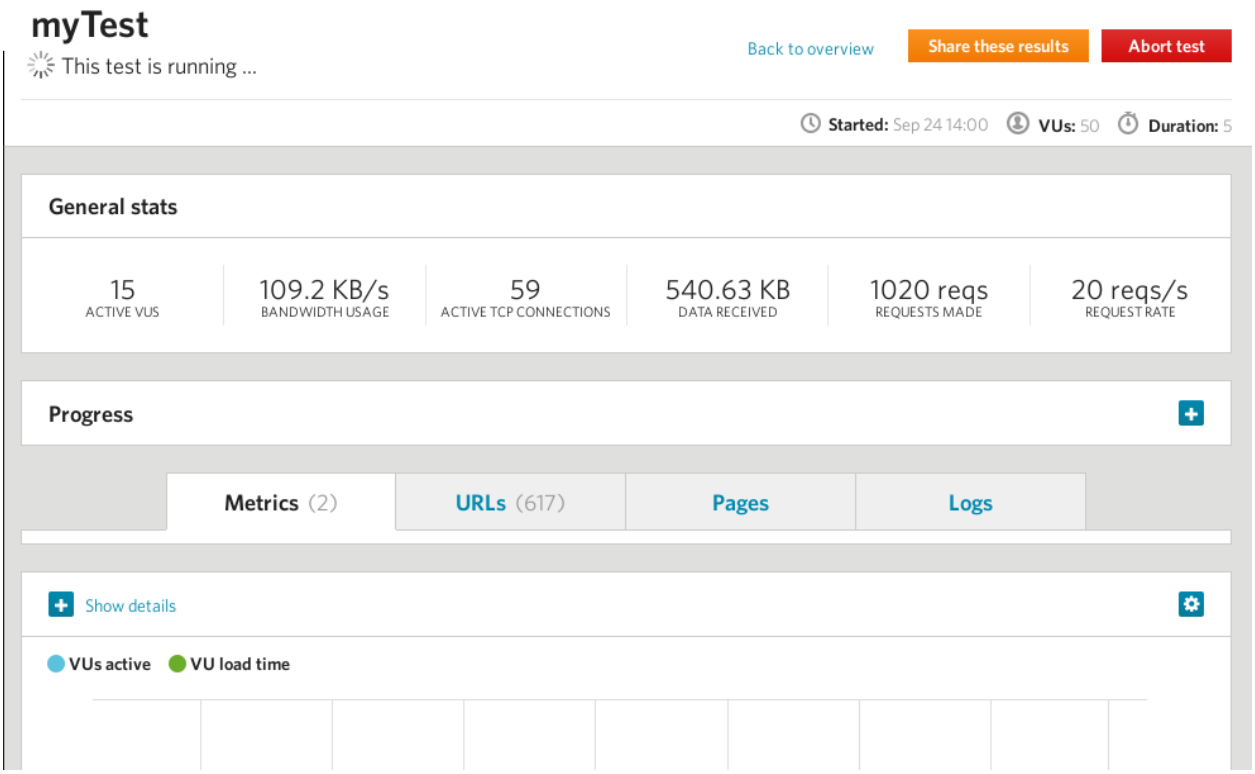


6. Click **Save changes.**

7. To begin the load test, click **Run test**.

8. As your test runs, view the statistics that are displayed during the test. You will analyze these test results in the next exercise.

Exercise 3.4: Load testing

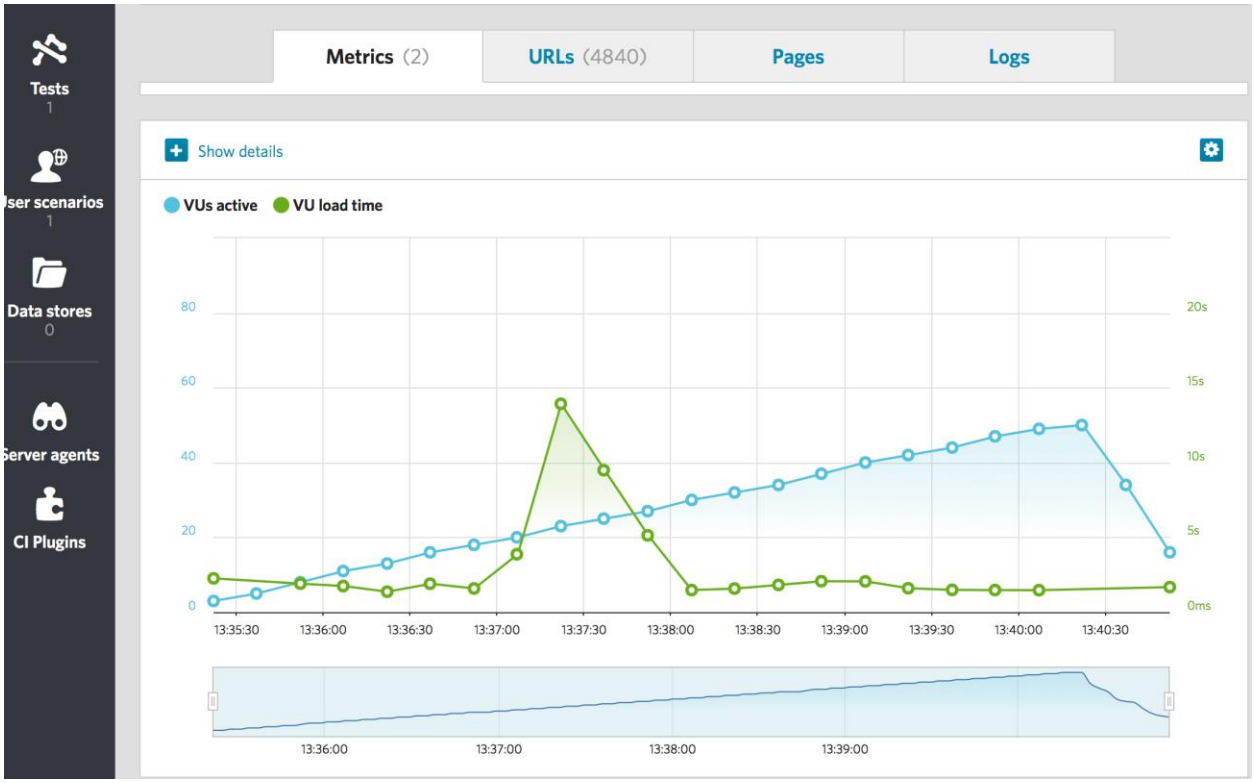


Exercise 3.4.6: Viewing the load impact test results and adding graphs

As the load impact test runs, data that is collected from load testing your application is generated. In this task, you will access and view some of this information.

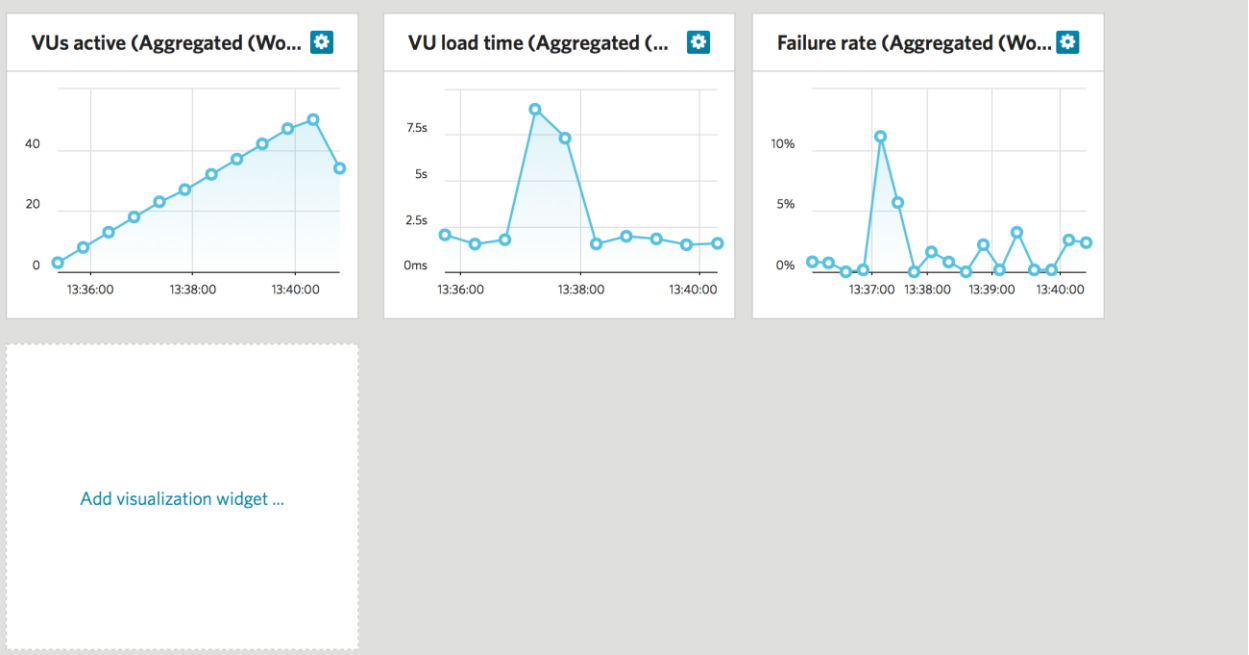
Complete the following steps:

1. Scroll through the Load Test page for the test that you just started and open the **Metrics** section. As the load impact runs, it generates graphs for your test. The first graph shows the number of active clients over time and the user load time. You can see whether the load time slows with the increased user load.



You can use the **Add visualization** widget to add more metrics to the same chart or to add more charts in addition to the default chart. For example, you can add test failure rates over time.

Exercise 3.4: Load testing



The Load Impact service provides only some of the server monitoring tools that you can use given the services you have bound to your application. You will use these analysis tools in the next section.

Note the following items:

- Configure the Auto-Scaling policy the same way as described in Exercise 3.2 for the instance used in that section
- Configure the Monitoring and Analytics service as described in Exercise 3.5 for the instance used in that section.
- View the advantages provided in the Metrics Statistics section by using Auto-Scaling and Monitoring and Analytics services for the application.
    - You can examine more closely the effects of the test on your auto-scaling policy rules that you first configured in Exercise 3.2.
    - The Monitoring and Analytics service configuration you used in Exercise 3.5 provides performance monitoring resource metrics for your Bluemix Java Liberty application. In this part of the service, you can identify potential issues with your application at run time.

Exercise 3.4.7: Sample test code for the load impact scenario

The following sample code shows a load impact sample test. Because your web application uses a URL that differs from the URL in the script, you must modify the `WEBSITE_URL` code by setting the variable to your own application's Bluemix URL.

```
-- Variable determines appropriate website URL to test.

-- EDIT this VARIABLE to test your application.

-- Ensure there is NOT a trailing slash at the end.

 local WEBSITE_URL = "http://replace-me.mybluemix.net"

-- User input we want to randomly generate.

 local KEY1 = string.char(math.random(32, 126))

 local KEY2 = string.char(math.random(32, 126))

 local KEY3 = string.char(math.random(32, 126))

 local VAL1 = math.random(1, 31415926)

 local VAL2 = math.random(1, 31415926)

 local VAL3 = math.random(1, 31415926)

-- Send PUT request to the app with random key-value pair.

http.request_batch({

{"GET", WEBSITE_URL .. "/ecaas?operation=put&key=" .. KEY1 .. "&value=" .. VAL1

.. "&encrypt=false", auto_decompress=true}

})

-- Simulate cache refresh by fetching all values.

http.request_batch({

{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY1 .. "&value=" .. VAL1

.. "&encrypt=false", auto_decompress=true}
```

```
})
-- Send second PUT request to the app with random key-value pair.
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=put&key=" .. KEY2 .. "&value=" .. VAL2
.. "&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY2 .. "&value=" .. VAL2
.. "&encrypt=false", auto_decompress=true}
})
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=put&key=" .. KEY3 .. "&value=" .. VAL3
.. "&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY3 .. "&value=" .. VAL3
.. "&encrypt=false", auto_decompress=true}
})
-- Send GET request to fetch the Value from Key. Then, refresh the data cache.
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=get&key=" .. KEY1 ..
"&value=&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY1 .. "&value=" .. VAL1
.. "&encrypt=false", auto_decompress=true}
})
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=get&key=" .. KEY2 ..
```

```
"&value=&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY2 .. "&value=" .. VAL2
.. "&encrypt=false", auto_decompress=true}
})
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=get&key=" .. KEY3 ..
"&value=&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY3 .. "&value=" .. VAL3
.. "&encrypt=false", auto_decompress=true}
})
-- Delete keys one through three, refreshing the cache after each request
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=delete&key=" .. KEY1 ..
"&value=&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=&value=&encrypt=false",
auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=delete&key=" .. KEY2 ..
"&value=&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=&value=&encrypt=false",
auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=delete&key=" .. KEY3 ..
"&value=&encrypt=false", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=&value=&encrypt=false",
```

```
auto_decompress=true}
})
-- Send an ENCRYPTED PUT request to the app with the random key-value pair.
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=put&key=" .. KEY1 .. "&value=" .. VAL1
.. "&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY1 .. "&value=" .. VAL1
.. "&encrypt=true", auto_decompress=true}
})
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=put&key=" .. KEY2 .. "&value=" .. VAL2
.. "&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY2 .. "&value=" .. VAL2
.. "&encrypt=true", auto_decompress=true}
})
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=put&key=" .. KEY3 .. "&value=" .. VAL3
.. "&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY3 .. "&value=" .. VAL3
.. "&encrypt=true", auto_decompress=true}
})
-- Send GET request for ENCRYPTED Value from Key. Then, refresh the data cache.
http.request_batch({
```

```
{"GET", WEBSITE_URL .. "/ecaas?operation=get&key=" .. KEY1 ..
"&value=&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY1 ..
"&value=aJY1vAbJJA2tCx2HN8QoQA%3D%3D&encrypt=true", auto_decompress=true}
})
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=get&key=" .. KEY2 ..
"&value=&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY2 ..
"&value=3o29aRTtQAPQBq1uW7zSkg%3D%3D&encrypt=true", auto_decompress=true}
})
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=get&key=" .. KEY3 ..
"&value=&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=" .. KEY3 ..
"&value=zQ7y%2Frb9pnRKfvxoPk%2B3cA%3D%3D&encrypt=true", auto_decompress=true}
})
-- Delete Keys one and two, leaving three. Refresh cache after each request.
http.request_batch({
{"GET", WEBSITE_URL .. "/ecaas?operation=delete&key=" .. KEY1 ..
"&value=&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=&value=&encrypt=true",
auto_decompress=true},
```

```
{"GET", WEBSITE_URL .. "/ecaas?operation=delete&key=" .. KEY2 ..
"&value=&encrypt=true", auto_decompress=true},
{"GET", WEBSITE_URL .. "/ecaas?operation=all&key=&value=&encrypt=true",
auto_decompress=true}
})
client.sleep(math.random(20,40))
```