Testing Framework [Beta]

Note: This platform offering is in the beta phase and is susceptible to significant changes in the forthcoming FDK releases.

The FDK includes a unit testing framework to help test serverless apps. Our framework is built using the popular mochaV5.

Typically, local testing of serverless apps is performed through simulated events. This is cumbersome and may not be recursive when there are feature additions to an app. The testing framework enables you to create automated unit tests that can run quickly, be replicated, and grow as the app code expands.

The framework enables you to write and maintain unit tests as part of the app files. Unit tests accelerate the review time of an app that you submit for publishing to the Freshworks Marketplace. This is because of,

- **Increased code coverage:** Automated tests written using the framework cover code paths that manual testing cannot invoke.

- **Ability to perform regression testing**: Maintainable and automated tests help sign off apps before you submit them to the Marketplace.

Prerequisites

- Node 10.x or later versions.
- FDK 6.2.7 or later versions.
- Working knowledge of:

    - **Test runners**: Tool or library that picks the suite of unit tests written, runs the tests based on certain settings, and logs the test results either on the console or in log files. The testing framework uses the test runner in-built with the Mocha framework.

    - **Assertion libraries**: Libraries that enable you to write assertions that validate test outcomes against certain conditions. Assertions eliminate the need for multiple

conditional statements to validate a test outcome. You can use any assertion library that meets your requirements.

- **Stubs**: Fake interfaces or classes that enable you to return pre-programmed values.

Specify dev dependencies

To use an assertion library,

1. Navigate to the **manifest.json** file in your app's root directory.
2. Specify the assertion library's name and version number as a devDependency. You can add all libraries and packages required for testing as devDependencies.

Sample manifest.json file

```
{
 "platform-version": "2.0",
 "product": {
   "freshdesk": {
     "location": {
       "ticket_sidebar": {
         "url": "template.html",
         "icon": "./resources/img/normal-github-64.svg"
       }
     }
   }
 },
 "devDependencies": {
   "chai": "4.2.0"
 }
}
```

Write Tests

Using the testing framework, you can write unit tests for a serverless app, similar to any generic unit test written using Mocha and an assertion library. The FDK offers two custom interfaces to help write unit tests.

1. **The stub interface**: Enables you to provide pre-programmed values that are used to stub specific objects to the app logic. You can stub the following objects (platform features):
    - Data Storage ($db)
    - Request Method ($request)
    - Schedules ($schedule)
    - generateTargetUrl()

2. **The invoke interface**: Invokes the serverless event for which the unit test is written.

Before you write tests, navigate to the app's root directory and create a **test** directory. To maintain an uncluttered test file, you can include the sample test payloads required to run all serverless events, as JSON files in a folder in the **test** directory. You can use the require() method to include a test payload to the test file.

All Mocha objects are invoked through the **this** keyword. The testing framework's interfaces are also invoked through the **this** keyword.

Stub interface

The testing framework uses Sinon.JS stubs to provide custom stub objects.

To stub an object that resolves with the specified value, use one of the following formats:

Note: In the following formats,

- `object` specifies the object that is stubbed. For example, $db, $request, and so on.
- `primitive` specifies the primitive parameter associated with the object. For example, get or set for $db and post for $request.

Syntax

```
this.stub('object').resolves('value');
```

Sample

```
const stub = this.stub('generateTargetUrl').resolves('http://randomurl.com/webhook');
```

Syntax

```
this.stub('object', 'primitive').resolves({key:value});
```

Sample

```
  const stubbedRequest = this.stub('$request', 'post').resolves({
    response:{
      url: 'http://randomurl.com/webhook'
    }
  });
```

To stub an object that rejects with the specified error, use one of the following formats:

Syntax

```
this.stub('object','primitive').rejects('error message');
```

Sample

```
  const StubbedRequest =  this.stub('$request', 'post').rejects(
    {
      error: 'Unable retrieve data'
    }
  );
```

Syntax

```
this.stub('object').rejects({error:'error message'});
```

Sample

```
  const stubbedGenerate = this.stub('generateTargetUrl').rejects({
    error:'unable to generate target url'
  });
```

To stub an object with an asynchronous function, use the following format:

Syntax

```
this.stub('object', 'primitive').callsFake(function(key, value){
 // Do Action

 // Return
});
```

Sample

```
// Stubbing a $request
   const stubbedRequest = this.stub('$request', 'post').resolves({
     response:{
       url: 'http://randomurl.com/webhook'
     }
   });

// Stubbing a $db object with callsFake
   const stubbedDB = this.stub('$db', 'set').callsFake(function(key, value) {

     // Asserts the key of the $db object to be saved
     expect(key).to.equal('githubWebhookId');
     // Asserts the response of the stubbed $request in the previous step
     expect(value.url).to.equal('http://randomurl.com/webhook');

     // Restores all stubs so that the objects can be stubbed later
     stubbedRequest.restore();
     stubbedDB.restore();
     return Promise.resolve()
   });
```

Sample unit tests

server/server.js

```
exports = {

 events: [
   { event: 'onTicketCreate', callback: 'onTicketCreateHandler' }
 ],

/**
*
* @param {object} args  Payload from the event
*/
onTicketCreateHandler: function (args) {

   var title = `${args.data.employee.first_name} ${args.data.employee.last_name}`;
   var email = args.data.employee.user_emails[0];
   var description = ` Please initiate issuance of laptop and other devices to the
${employeeName}`


$request.post(`https://${args.freshservice_subdomain}.freshservice.com/api/v2/tickets`
, {
     headers: {
       Authorization: "Basic <%= encode(iparam.freshservice_api_key)%>",
       "Content-Type": "application/json;charset=utf-8"
     },
      body: JSON.stringify({
       description: `${description}`,
       email: `${email}`,
       priority: 1,
       status: 2,
       subject: `${title}`
     })
   }).then(function () {
     console.info('Successfully created ticket');
   }).catch(
```

```
      function (error) {
     console.error('Unable to create ticket');
     console.error(error);
   });
 }
};
```

test/server.js

```
'use strict';

const expect = require('chai').expect;
const onTicketCreateArg =  require(`../server/test_data/onTicketCreate.json`);

describe('App Events Spec', function() {
 it.only('Checks app success workflow', function() {
   const stubbedRequest =  this.stub('$request', 'post').callsFake((url, payload) => {
     stubbedRequest.restore();
     return Promise.resolve();
   });

   this.invoke('onTicketCreate',onTicketCreateArg);

 });
 it('checks app failure case',function() {
    const stubbedRequest =  this.stub('$request', 'post').rejects({
     response: {
       url: 'Unable to create ticket'
     }
   });
   this.invoke('onEmployeeCreate',onEmployeeCreateArg)
   stubbedRequest.restore();
  });
});
```

Invoke interface

To call serverless events by using the invoke interface, use the following format:

Note: In the following format,

- `event` specifies the name of the event in the **server.js** file that is to be invoked to test the unit test written.
  Possible values: **onAppInstall**, **onAppUninstall**, **onExternalEvent**, **onScheduledEvent**, or any valid product event configured in the **server.js** file.
- `payload` specifies the payload that is passed to the event. The payload object does not contain the iparams object. The testing framework adds the iparams object to the payload that is sent to the event handler.

Syntax

```
// Include payload to this file
const payload = require('./args/payload')

// Invoke the app event
this.invoke('event',payload);
```

Sample

```
const appInstallArg = require('./args/appInstallArg')

this.invoke('onAppInstall', appInstallArg);
```

Run tests

1. Navigate to the app's root directory and create a directory called **test**, if you haven't created one when writing tests.

2. Add the test files that contain the units test written, to the **test** directory.
3. From the command line, navigate to the app's root directory and run the following command:
   $ fdk test