











Next-Gen B2C Service Extensibility: Replace Custom Scripts APIs with OCI API Gateway

Learn how to leverage the OCI API Gateway and OCI Functions to build a better custom API solution for B2C Service

September, 2020 | Version 1.01
Copyright © 2020, Oracle and/or its affiliates
Confidential – Public

TABLE OF CONTENTS

Introduction	3
Use Case	3
Design Overview.....	3
Process Diagram.....	4
Technical Diagram	5
Implementation	6
Before You Begin ←.....	6
SMS Payload 	6
The Function 	7
Getting Integration Secrets 	7
Consuming the Webhook 	8
Integrating with B2C Service 	8
Test Your Function 	10
Publish to OCI 	11
Configure OCI API Gateway 	11
Create the Gateway 	11
Test the Gateway 	15
Conclusion	16

INTRODUCTION

Oracle B2C Service extensibility is improving by leveraging capabilities of Oracle Cloud Infrastructure (OCI). The suite of tools that OCI provides allows you to build new customizations in much more flexible ways than previously possible with B2C Service extensibility. In this guide, we will explore using the [OCI API Gateway](#) and [OCI Functions](#) as a replacement for providing custom API endpoints using custom scripts or Customer Portal.

B2C Service customers need custom API endpoints to extend B2C Service capabilities in circumstances where an external service needs to call B2C Service but cannot use the REST APIs. Typically, these are external webhooks that need to call B2C Service such as those provided by SMS and telephony vendors.

Historically, customers have used `PHP` scripts to provide custom API endpoints in B2C Service. These scripts, while convenient to implement on the B2C Service platform, require the implementer be aware of the security and scaling implications that are introduced with these types of scripts. It has never been a best practice to implement a customization like this, however, alternatives were also difficult to implement. Until now!

The OCI API Gateway service provides solutions to all of the issues that a custom API implemented in custom scripts introduce, namely authentication & authorization, rate limiting, scalability, and monitoring.

Use Case

For this post, we will use an example organization that needs to implement an API that can receive a webhook from an SMS service for the creation of service incidents via SMS. To keep the example simple, any inbound SMS will create a new incident, but the business logic could be updated as needed to account for incident updates or other B2C Service data operations.

Design Overview

The solution is comprised of the following components from OCI:

- [API Gateway](#) allows us to expose a secure HTTP endpoint over the public internet with rate limiting and CORS applied to it.
- [Oracle Functions](#) allows development, deployment, and execution of applications that implement business logic without worrying about the infrastructure where it will execute. The API Gateway will trigger a function which will consume the SMS payload and integrate with B2C Service via the REST API.
- [OCI Vault](#) allows us to store the credentials for a REST API service user for B2C Service so that our function can integrate with B2C Service while retrieving credentials securely at runtime.
- The [OCI CLI](#) is a toolkit that you install on your computer to interact with OCI via the command line on your local machine.

Be sure to review the pricing documentation for OCI to plan the cost for your OCI usage. You can use the [cost estimator](#) to estimate your usage costs.

Process Diagram

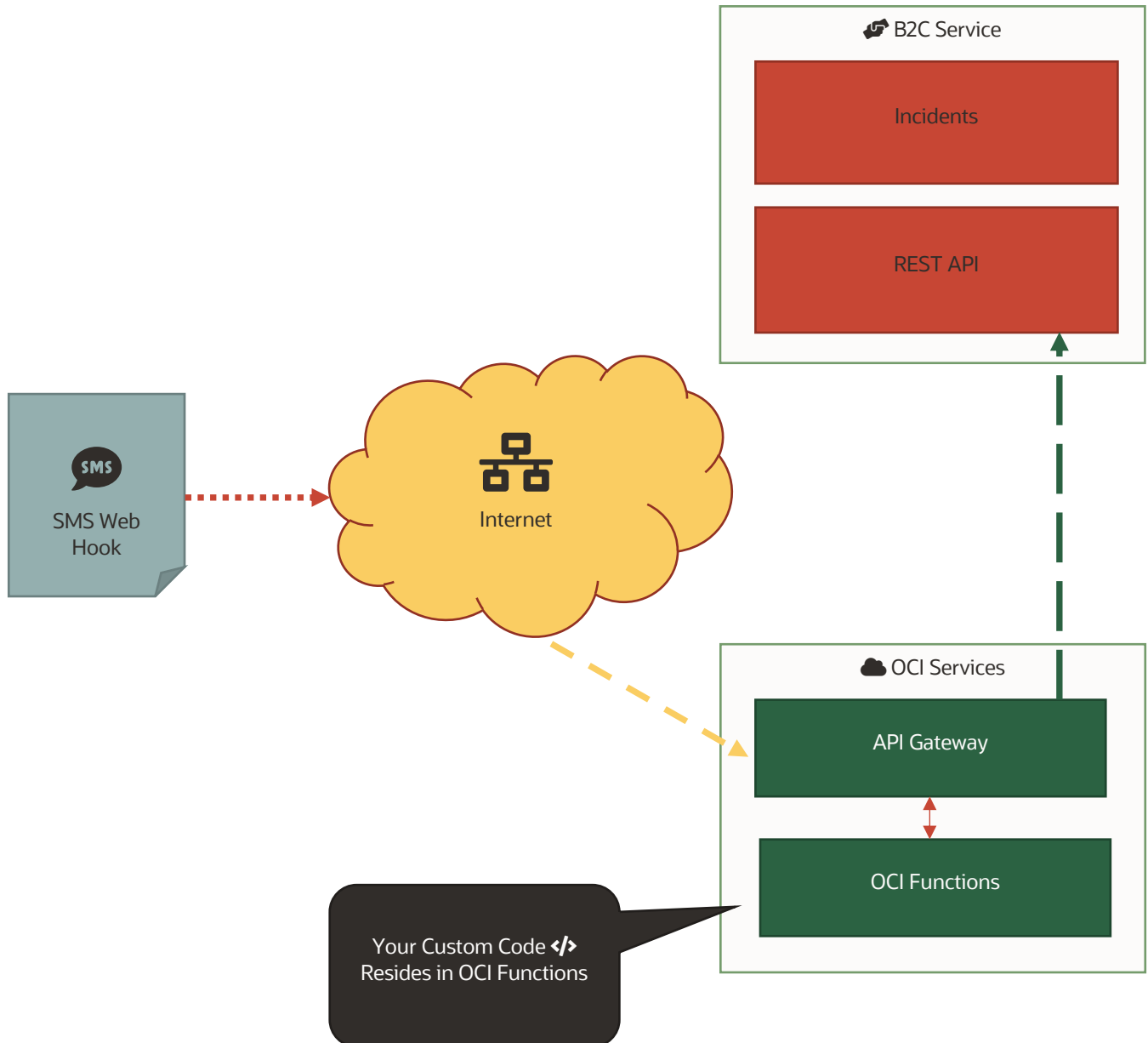
The following diagram demonstrates the end-to-end flow that this example process will demonstrate:



Technical Diagram

The following diagram demonstrates where the different components in this solution reside and how they communicate with each other.

1. SMS triggers a webhook from SMS service to OCI API Gateway
2. OCI API Gateway authenticates the webhook request and passes the payload to an OCI Function
3. The OCI Function transforms the SMS payload to an incident creation request for B2C Service
4. The OCI Function creates an incident in B2C Service via the REST API



IMPLEMENTATION

The implementation for this effort will be delivered entirely using tools in Oracle Cloud Infrastructure; customizing B2C Service is not required for this type of solution! The only details needed from B2C Service will be a service account for integration with the B2C Service REST APIs.

The implementation will be completed in the following steps:

1. OCI tenancy configuration. This step needs to be completed outside the scope of this guide. Your tenancy needs to be configured with a public subnet that our OCI API Gateway can use to expose a public endpoint and for our function to make public REST API calls. It also needs to be configured with the tenancy and compartment policies that allow the API Gateway to call functions and for functions to access secrets.
2. Development of a function that will take an SMS payload and create a B2C Service incident from the information in that payload
3. Configuration of an application in IDCS to manage the permission model that needs to be applied to our API
4. Configuration of an API Gateway that will expose an HTTP endpoint for our SMS vendor to post data to.

Before You Begin ←

Be sure that you have the following items available to you before you begin:

1. B2C Service account username and password for integration with the REST API
2. An OCI tenancy with a compartment identified for this implementation
3. Permissions in your compartment to manage the implementation for OCI API Gateway, OCI Functions, OCI VCN, Vault/Secrets and any other required OCI capabilities
4. A SMS service provider

SMS Payload

In order to deliver a generic example in this guide, the following payload will be the assumed payload provided by the SMS provider. You will need to update the expected payload in your implementation based on the information provided by your SMS vendor.

```
{
  "account_sid": "ACXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "api_version": "2010-04-01",
  "body": "McAvoy or Stewart? These timelines can get so confusing.",
  "date_created": "Thu, 30 Jul 2015 20:12:31 +0000",
  "date_sent": "Thu, 30 Jul 2015 20:12:33 +0000",
  "date_updated": "Thu, 30 Jul 2015 20:12:33 +0000",
  "direction": "outbound-api",
  "error_code": null,
  "error_message": null,
  "from": "+15017122661",
  "messaging_service_sid": "MGXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "num_media": "0",
  "num_segments": "1",
  "price": null,
  "price_unit": null,
  "sid": "SMXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "status": "sent",
  "subresource_uris": {
```

```

    "media": "/2010-04-
01/Accounts/ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Messages/SMXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Me
dia.json"
  },
  "to": "+15558675310",
  "uri": "/2010-04-
01/Accounts/ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Messages/SMXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.js
on"
}

```

The Function

This guide comes with a python file that provides the business logic that will consume the webhook payload above and create a B2C Service incident. It can act as the foundation of a similar implementation based on your own business requirements.

Note: Prerequisites to this section are reviewing the [F\(n\) project documentation](#) for function development and local testing and [OCI Functions](#) for cloud deployment. This section will not go into the implementation process of a function since the product documentation supplies that information already.

Getting Integration Secrets

One of the benefits of using OCI Functions is that you can manage environment variables for your functions so that you do not need to hard-code variables like integration details. Also, there is the added benefit of storing secret data in the secure OCI vault service so that environment variables that need to be secured, such as usernames and passwords, are completely isolated from the function script.

The example below demonstrates getting the required environment variables for integration with a B2C Service site. The variables are stored at the function's application so that they can be shared across all functions in the application.

When the function is run in OCI, the script expects that the username and password variables be retrieved from Vault, so we use the OCID of the secrets that represent those variables from setting up the secrets (see [OCI Vault](#) for more information). When run locally (for testing), the function expects that the variables are available as regular function configurations.

```

# Params that will be assigned based on where this function is running
username = None
password = None

# Get environment vars from F(n)
hostname = os.getenv("b2cservice_hostname")

# If the function is running in OCI, then retrieve secrets from vault
if 'oci' in app_id:
    #logging.getLogger().debug("Using OCI config settings")

    username_ocid = os.getenv("b2cservice_username_ocid")
    password_ocid = os.getenv("b2cservice_password_ocid")

    # By default this will hit the auth service in the region the instance is
running.
    signer = oci.auth.signers.get_resource_principals_signer()

    # Get instance principal context

```

```

secret_client = oci.secrets.SecretsClient(config={}, signer=signer)

# Get B2CService Credentials from Vault
username = read_secret_value(secret_client, username_ocid)
password = read_secret_value(secret_client, password_ocid)
# else, if running locally, then use env params
else:
    #logging.getLogger().debug("Using local config settings")

    username = os.getenv("b2cservice_username")
    password = os.getenv("b2cservice_password")

```

Consuming the Webhook 📡

Load the data into a variable from the function's data parameter. This process is similar for any function that will have data posted to it.

```

# Parse Webhook Body
try:
    webhook_data_str = data.getvalue().decode('utf-8')
    #logging.getLogger().debug(webhook_data_str)

    webhook_data = json.loads(webhook_data_str)

```

Integrating with B2C Service 📠

Next, we will call a few convenience methods in sequence to perform the following steps:

1. Determine if a contact exists that contains the phone number that was the source of the SMS. If not, then a new contact is created with a generic first and last name since no other contact information exists at this point.
2. Create an incident from the SMS. The previously retrieved/created contact will be associated to the incident as the primary contact, the subject will be the body of the SMS, and a thread will be created with the full payload of the inbound webhook for reference to the original data.

```

if 'body' in webhook_data:
    sms_body = webhook_data['body']
    sms_from = webhook_data['from']
    sms_to = webhook_data['to']

    # Check to see if there is a contact with the "to" phone number
    contact_id = get_contact_by_mobile(osvc_client, sms_from)

    # If no contact was returned, then create one
    if contact_id == None:
        contact_id = create_contact_from_sms(osvc_client, sms_from)

    # Create an incident from the inbound thread
    new_incident = create_incident_from_sms(osvc_client, contact_id, webhook_data)

```

The first call to `get_contact_by_mobile` will query the contact by phone number. If there is no result, then `None` is returned and we know that we need to create a new contact.


```
def get_contact_by_mobile(osvc_client, number) :
    contact_results = osvc_client.get("/services/rest/connect/latest/contacts?q=phones.number={}".format(number))

    logging.getLogger().debug(json.dumps(contact_results, indent=3))

    if 'items' in contact_results and len(contact_results['items']) > 0 and 'id' in contact_results['items'][0]:
        return contact_results['items'][0]['id']

    return None
```

If required, we call `create_contact_from_sms`, which will create a new contact record with the phone number used to generate the SMS message.

```
def create_contact_from_sms(osvc_client, number):

    contact = {
        "name": {
            "first": "SMS",
            "last": "User"
        }
    }

    create_result = osvc_client.post("/services/rest/connect/latest/contacts",
    json.dumps(contact))

    if id in create_result and create_result['id'] != None:
        contact_id = create_result['id']

    new_phone = {
        "number": number,
        "phoneType": {
            "lookupName": "Mobile Phone"
        }
    }

    osvc_client.post("/services/rest/connect/latest/contacts/{}/phones".format(contact_id),
    json.dumps(new_phone))

    return contact_id
else:
    raise Exception(create_result)
```

Lastly, we call `create_incident_from_sms` passing in the contact and SMS data to create the incident and incident thread as mentioned above.

```
def create_incident_from_sms(osvc_client, contact_id, webhook_data):
    subject = webhook_data['body']
```

```

new_incident = {
    "subject": subject,
    "primaryContact": {
        "id": contact_id
    }
}

create_result = osv_client.post("/services/rest/connect/latest/incidents",
json.dumps(new_incident))

if 'id' in create_result and create_result['id'] > 0:
    incident_id = create_result['id']

    # save the SMS body as a note to the incident for record keeping
    new_thread = {
        "text": json.dumps(webhook_data, indent=2),
        "entryType": {
            "lookupName": "Note"
        },
        "contentType": {
            "lookupName": "text/plain"
        }
    }

    logging.getLogger().debug(json.dumps(new_thread, indent=3))

osv_client.post("/services/rest/connect/latest/incidents/{}/threads".format(incident_id),
json.dumps(new_thread))

    return create_result
else:
    logging.getLogger().error("Could not get incident ID from create results")

    raise Exception(create_result)

```

Test Your Function

Now that we understand the business logic of our function's script, we can test it locally using the F(n) CLI. This document does not explore how to deploy a function to the local F(n) service, but that is covered by the F(n) documentation. In summary, follow these steps:

1. Create an application for your function using the F(n) CLI.
2. Add any application-wide configurations that are applicable (likely the integration variables for OSvC as those would be shared across all of the application's functions that need to integrate to B2C Service).
3. Create the function using the ``fn init`` command.
4. Copy the ``func.py`` and ``osvc.py`` files from this example into your function's directory.
5. Deploy the function locally.
6. Use the shell to pipe a demo payload to your function while you invoke it.

Assuming you have configured your function properly, you should see a success message once your functions finishes the invoke routine. And, there will be a new incident in B2C Service where you can verify that the script worked as expected.

Request:

```
echo -n '{"account_sid":"ACXXXXXXXXXXXXXXXXXXXXXXXXXXXX","api_version":"2010-04-01","body":"McAvoy or Stewart? These timelines can get so confusing.","date_created":"Thu, 30 Jul 2015 20:12:31 +0000","date_sent":"Thu, 30 Jul 2015 20:12:33 +0000","date_updated":"Thu, 30 Jul 2015 20:12:33 +0000","direction":"outbound-api","error_code":null,"error_message":null,"from":"+13365550622","messaging_service_sid":"MGXXXXXXXXXXXXXXXXXXXXXXXXXXXX","num_media":"0","num_segments":"1","price":null,"price_unit":null,"sid":"SMXXXXXXXXXXXXXXXXXXXXXXXXXXXX","status":"sent","subresource_uris":{"media":"/2010-04-01/Accounts/ACXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Messages/SMXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Media.json"},"to":"+15558675310","uri":"/2010-04-01/Accounts/ACXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Messages/SMXXXXXXXXXXXXXXXXXXXXXXXXXXXX.json"}' | fn invoke osv-webhooks sms-to-incident
```

Response:

```
HTTP/1.1 200 OK
Content-Length: 22
Content-Type: application/json
Date: Tue, 15 Sep 2020 14:49:00 GMT
Fn-Call-Id: 01EJ93TYPANG8G00GZJ0000021
Fn-Fdk-Version: fdk-python/0.1.18

{"message": "success"}
{"message": "success"}
```

Publish to OCI

Publishing a function to OCI is like publishing it locally, but there are a few differences that the documentation outlines as you move from your local testing to OCI. Namely, you will need to account for the required attributes, such as your function's VCN subnet attribute, that need to be added when deploying to OCI.

Follow the processes outlined in the [OCI Functions documentation](#) to deploy your application and function to OCI. Be sure to also setup the Vault secrets for the B2C Service Username and Password as the script expects those variables to be secured by Vault.

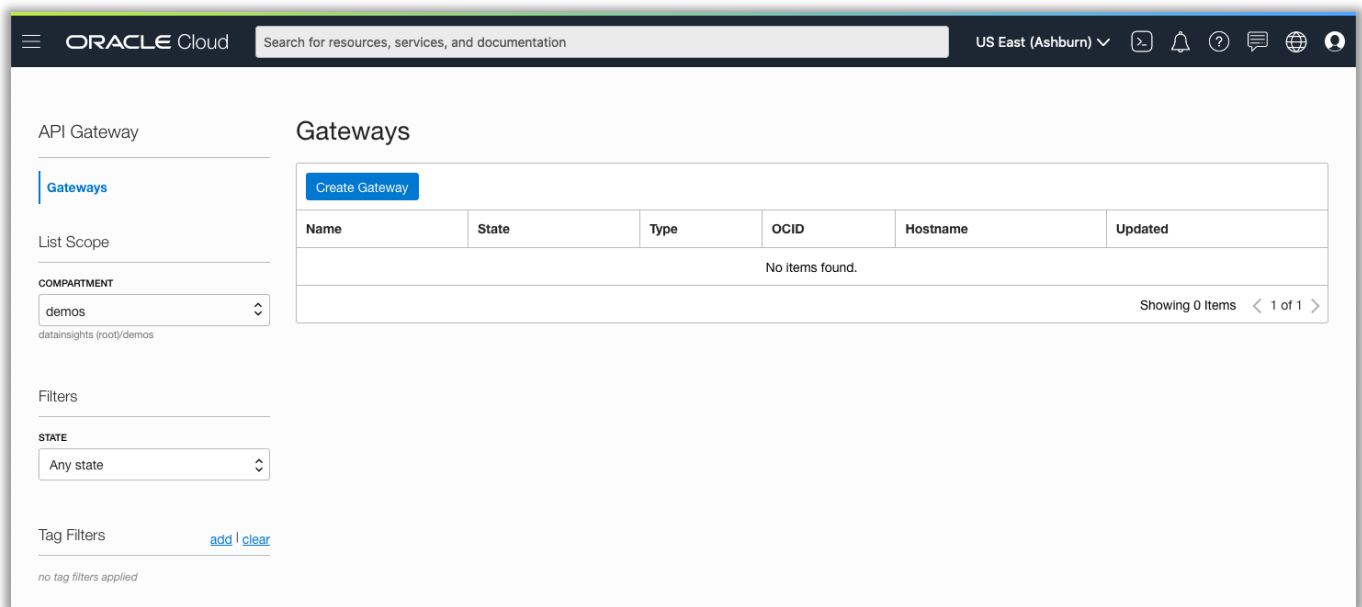
Testing your function in OCI may still be performed via the command line tools in order to verify that it is working as expected. You only need to start testing HTTP posts after the next section is complete.

Configure OCI API Gateway

It's time to create and expose an HTTP endpoint that a webhook can call in order to post data to your newly created function. We will walk through the high-level configuration steps in this guide, but refer to the [API Gateway documentation](#) for more detailed instructions for implementation.

Create the Gateway

Navigate to Developer Services > API Gateway in the OCI console.



Ensure that you are in the same compartment as your function. Click “Create Gateway” to create a new API Gateway in the same compartment as our function. Then, fill out the fields in the form that is presented. You will need to be sure to use a public subnet so that your gateway is exposed to the public internet.

The gateway may take a few minutes to be created.

Once the gateway is created, click on the “Deployments” link under “Resources” on the left side of the page. Then, click “Create Deployment”.

API Gateway » Gateways » Gateway Details

osvc-webhooks-gateway

ACTIVE

Gateway Information | Tags

OCID: ...n2tpze7zna [Show](#) [Copy](#)
 Type: Public
 Created: Mon, Aug 31, 2020, 15:11:52 UTC
 Updated: Mon, Aug 31, 2020, 15:12:31 UTC

Compartment: datainsights (root)/demos
 Subnet: [demo-subnet](#)
 Hostname: ...oci.customer-oci.com [Show](#) [Copy](#)

Resources

- Metrics
- Deployments**
- Work Requests

Filters

STATE

Any state

Create Deployment

Name	Path Prefix	State	Endpoint	Deployed
No items found.				

Showing 0 Items < 1 of 1 >

The next few steps will determine how your exposed REST endpoints will work. Use best-practices or required implementation paradigms for CORS support, etc. established by your organization. And, ensure that you are using an authorization mechanism so that your endpoint is not freely exposed to the public internet.

Create the basic information first. The “Path Prefix” field is the root of your API. You should name it something common to all the endpoints that will be below it.

Create Deployment

1 Basic Information
 2 Routes
 3 Review

From Scratch
 Deploy a custom API using the provided form

Upload an existing Deployment API
 Upload a JSON file containing the API

Basic Information

NAME
 osvc-webhooks-deployment

PATH PREFIX ⓘ
 /osvc-webhooks

COMPARTMENT
 demos
 datainsights (root)/demos

Setup an authentication type to secure your endpoint. In the case of an SMS provider, it's likely that you would implement a custom authorizer function to verify the sending IP address in order to validate the request; many SMS providers don't directly authenticate with webhook endpoints. In cases like this, the ability to leverage custom functions means that you can create any mechanism required to authenticate based on your integrator's capabilities. Of course, you can use an identify provider, like IDCS, for OAuth, JWT, and SAML authentication solutions.

ORACLE Cloud Search for resources, services, and documentation US East (Ashburn)

Create Deployment

1 Basic Information
2 Routes
3 Review

From Scratch
Deploy a custom API using the pr

Basic Information

NAME
osvc-webhooks-deployment

PATH PREFIX ⓘ
/osvc-webhooks

COMPARTMENT
demos
datainsights (root)/demos

Authentication Policy

AUTHENTICATION TYPE
Custom

APPLICATION IN AR ⓘ (CHANGE COMPARTMENT)
ar-accelerator

FUNCTION NAME ⓘ
api-auth

AUTHENTICATION TOKEN ⓘ
Header

HEADER NAME ⓘ
x-api-header

☐ ENABLE ANONYMOUS ACCESS ⓘ

Gateway osvc-webhooks-gateway is being created

Gateway osvc-webhooks-gateway was successfully created

It's also a good idea to setup CORS and rate limiting policies as per the requirements of your implementation.

Click “next” and review your API Gateway deployment. Then, click “create”.

Your changes will deploy within your gateway. It may take a few minutes to provision.

ORACLE Cloud Search for resources, services, and documentation US East (Ashburn)

API Gateway » Gateways » Gateway Details

API osvc-webhooks is being deployed

osvc-webhooks-gateway

Edit Move Resource Add Tags Delete

Gateway Information Tags

OCID: ...n2tpze7zna Show Copy
Type: Public
Created: Mon, Aug 31, 2020, 15:11:52 UTC
Updated: Mon, Aug 31, 2020, 15:12:31 UTC

Compartment: datainsights (root)/demos
Subnet: demo-subnet
Hostname: ...oci.customer-oci.com Show Copy

Resources

Metrics
Deployments
Work Requests

Filters

STATE
Any state

Scope

COMPARTMENT ⓘ
demos
datainsights (root)/demos

Deployments

Create Deployment

Name	Path Prefix	State	Endpoint	Deployed
osvc-webhooks	/webhooks	Creating	...m/webhooks Show Copy	Mon, Aug 31, 2020, 21:04:22 UTC

Showing 1 item < 1 of 1 >

Test the Gateway

Once deployed, it's time to test your API call from the public internet. We can use cURL from our shell to test.

Note: You may need to update your authorizer function to allow requests from your IP address or the proper parameters based on how that authorizer function was implemented.

```
curl -X "POST" "https://<gatewayid>.apigateway.us-ashburn-1.oci.customer-
oci.com/webhooks/sms-to-incident" \
  -H 'Content-Type: application/json; charset=utf-8' \
  -d '${
    "account_sid": "ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "api_version": "2010-04-01",
    "body": "McAvoy or Stewart? These timelines can get so confusing.",
    "date_created": "Thu, 30 Jul 2015 20:12:31 +0000",
    "date_sent": "Thu, 30 Jul 2015 20:12:33 +0000",
    "date_updated": "Thu, 30 Jul 2015 20:12:33 +0000",
    "direction": "outbound-api",
    "error_code": null,
    "error_message": null,
    "from": "+13365550652",
    "messaging_service_sid": "MGXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "num_media": "0",
    "num_segments": "1",
    "price": null,
    "price_unit": null,
    "sid": "SMXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
    "status": "sent",
    "subresource_uris": {
      "media": "/2010-04-
01/Accounts/ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Messages/SMXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Me
dia.json"
    },
    "to": "+15558675310",
    "uri": "/2010-04-
01/Accounts/ACXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/Messages/SMXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.js
on"
  }'
```

If your configuration is correct, then you should see a reply from your function that the result was a success! Next, check to ensure that your test created an incident in B2C Service. If so, then you should be ready to connect your SMS provider's webhooks to your new HTTP endpoint.



CONCLUSION

You should now be able to create a custom HTTP endpoint using OCI API Gateway for custom API endpoints that you need for B2C Service. This approach allows you to build and scale your custom API needs in a secure and supported fashion.

CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com.
Outside North America, find your local office at oracle.com/contact.

 blogs.oracle.com

 facebook.com/oracle

 twitter.com/oracle

Copyright © 2020, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

This device has not been authorized as required by the rules of the Federal Communications Commission. This device is not, and may not be, offered for sale or lease, or sold or leased, until authorization is obtained.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0120

Disclaimer: This document is for informational purposes. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described in this document may change and remains at the sole discretion of Oracle Corporation.

