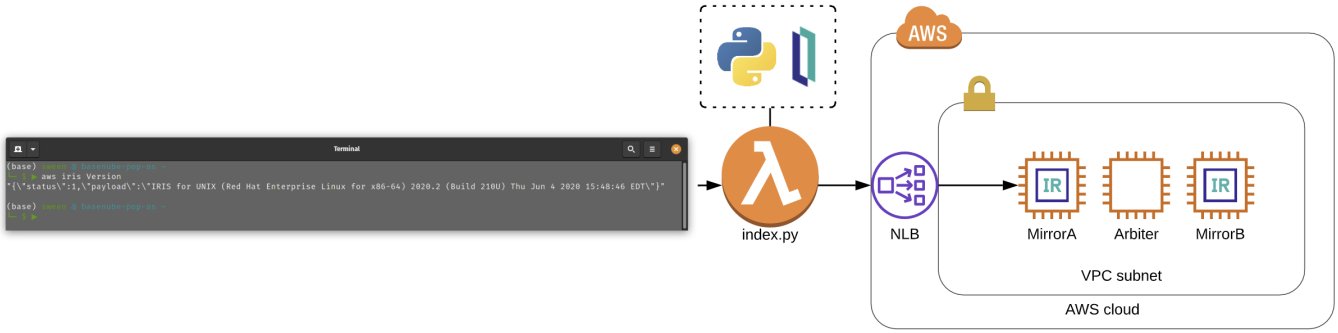
記事
Toshihiko Minamoto · 2021年8月24日 8m read

AWS Lambda の IRIS Python Native API IRIS

ソリューションを Amazon Web Services エコシステム、サーバーレスアプリケーション、または boto3 を使用した Python スクリプトにスムーズに統合する方法を探している場合は、IRIS Python Native API を使用するのが最適です。 IRIS で何かを取得したり設定したりしてアプリケーションに素晴らしい機能を備える必要があるまで、本番の実装を大々的に構築する必要はありません。したがって、誰かにとって重要なものまたはあなた以外にはまったく重要でないもの（これも同じくらい重要なことです）を構築できるようにこの記事が役に立てればと思います。



これを実装する口実が必要であれば

- OAUTH2 ワークフローを実装する FHIR(R) ベースソリューションで Patient ID コンテキストを調べ、SHMAHT のトークンに挿入するために Pre Token Generation Trigger を使用する必要がある。
- インスタンスタイプ、ノードグループ、Toaster、または ECS クラスタに応じてプロビジョニング後の IRIS チューニング設定を行い、リングゼロで IRIS を実行したい。
- AWS CLI でシェルを終了することなく IRIS を管理できるスキルを Zoom で家族や友人に披露したい。

ポイント

ここでは、IRIS と対話できる AWS lambda 関数を示し、それをプロビジョニングする方法とさまざまなキャパシティで操作する方法に
ついての例をいくつか説明します。これに関する対話を持つことで、**物事をより簡単に実現できる pip を公開**できるようになることを願っています。

お急ぎですか？？？？

Stream をご覧ください。

https://www.youtube.com/embed/mA0VzKOYhBk
*[これは埋め込みリンクですが、あなたはサイト上の埋め込みコンテンツへのアクセスに必要な Cookie を拒否しているため、それを直接表示することはできません。埋め込みコンテンツを表示するには、**Cookie 設定**ですべての Cookie を受け入れる必要があります。]*

そうでない方は…

このお楽しみに参加するには、現在のフライトプランから項目をいくつか削除する必要があります。

ネットワークの構築

IRIS の実行 OS は問いませんが、きっかり 2 つのサブネットを使用する AWS VPC で実行しており、IRIS を実行しているスーパーサーバーへのアクセスを許可するセキュリティグループが必要です。懐かしさと InterSystems が時間を掛けて IANA に登録したという単なる事実からポート 1972 を使用していますが、時間を掛けずに新しいポートを /etc/services に追加する場合は、新しいマットレスのタグを引きちぎるのと同じくらいに大変な思いをすることでしょう。私たちの場合、AWS v2 ネットワークロードバランサーに適切なヘルスチェックを備えた ec2 インスタンスのミラーセットです。

インポートしたサンプルのクラス

どうにかして、このリポジトリのルートにクラスを作成して IRIS インスタンスの %SYS ネームスペースにインポートすることができました。 以下は、上記の出力を駆動するクラスの例です。 ここでクラスをインポートする必要がある理由に疑問を持っている場合は、以下の注意事項をご覧ください。Python で使用するラッパークラスをプロビジョニングすることがアプローチとして推奨されています。

> ドキュメントから抜粋した注意事項: これらのメソッドはクラスライブラリに定義されている InterSystems クラスと使用することもできますが、メソッドをユーザー定義クラスまたはルーチン内から間接的に呼び出す方法をベストプラクティスとしています。
> 多くのクラスメソッドはステータスコードのみを返し、実際の結果は引数に返されます（Native API はアクセスできません）。 システム定義関数（ObjectScript リファレンスの ObjectScript 関数に記載された関数）を直接呼び出すことはできません。

サンプルのクラス:

```
Class ZDEMO.IRIS.Lambda.Operations Extends %Persistent
{
  ClassMethod Version() As %String
  {
   Set tSC = 0

   Set tVersion = $ZV
   if ( tVersion '="" ) { set tSC = $$$OK }

   Set jsonret = {}
   Set jsonret.status = tSC
   Set jsonret.payload = tVersion

   Quit jsonret.%ToJSON()
  }
}
```

ここでは慣習に基づいて作業し、常に JSON オブジェクトをレスポンスとして返すことにしました。こうすればステータスを返すことも可能で、参照によって何かを返せばギャップを埋められるかもしれません。

AWS アクセス

世界を変えようとしている Lambda 関数をプロビジョニングして呼び出せるように、IAM アクセスキーを取得してください。

飛行前チェック:

```
IRIS            [ $$$OK ]
VPC             [ $$$OK ]
Subnets         [ $$$OK ]
Security Group  [ $$$OK ]
IAM Access      [ $$$OK ]
Imported Class  [ $$$OK ]
```

$$$OK, **レスゴー**。

Lambda 関数で使用する IRIS Native Python API をパッキング

AWS Lambda 関数は Linux の Boxen 上で実行するため、これは、特に Linux のみの場合に pip パッケージであれば素晴らしい部分です。 このコミットの時点では、API は pip 経由で使用できませんが、リソースは豊富に備えているため、独自のものを用意することができます。

```
mkdir iris_native_lambda
cd iris_native_lambda
wget https://github.com/intersystems/quickstarts-python/raw/master/Solutions/nativeAP
I_wheel/irisnative-1.0.0-cp34-abi3-linux_x86_64.whl
unzip nativeAPI_wheel/irisnative-1.0.0-cp34-abi3-linux_x86_64.whl
```

connection.config を作成します。

例: connection.config

ハンドラー index.py を作成するか、demo github リポジトリの examples フォルダにあるものを使用します。デモバージョンでは、IRIS 接続情報に環境変数と外部ファイルの両方が使用されていることに注意してください。

例:
index.py

使用できるようどのように zip 圧縮するか。

```
zip -r9 ../iris_native_lambda.zip *
```

S3 バケットを作成し、関数の zip をそれにアップロードします。

```
cd ..
aws s3 mb s3://iris-native-bucket
s3 sync iris_native_lambda.zip s3://iris-native-bucket
```

これで、AWS Lambda 関数として使用する API とハンドラーのパッケージ化は完了です。

では、コンソールをクリックして関数を作成するか、Cloudformation などを使用して作成します。

```
  IRISAPIFunction:
    Type: "AWS::Lambda::Function"
    DependsOn:
      - IRISSG
      - VPC
    Properties:
      Environment:
        Variables:
          IRISHOST: "172.31.0.10"
          IRISPORT: "1972"
          NAMESPACE: "%SYS"
          USERNAME: "intersystems"
          PASSWORD: "lovetheyneighbor"
      Code:
        S3Bucket: iris-native-bucket
        S3Key: iris_native_lambda.zip
      Description: "IRIS Native Python API Function"
      FunctionName: iris-native-lambda
      Handler: "index.lambda_handler"
      MemorySize: 128
      Role: "arn:aws:iam::8675309:role/BeKindtoOneAnother"
      Runtime: "python3.7"
      Timeout: 30
      VpcConfig:
        SubnetIds:
          - !GetAtt
            - SubnetPrivate1
            - Outputs.SubnetId
          - !GetAtt
            - SubnetPrivate2
            - Outputs.SubnetId
        SecurityGroupIds:
          - !Ref IRISSG
```

大変な作業でしたが、これで Python でラムダ関数を使って IRI
を呼び出せるようになりました。世界が大きく変わります。

実行！

上記の実装方法では、関数は、再利用できるように構造化されたイベントオブジェクトに渡すことが期待されています。この考え方は以下のイベントオブジェクトの例で確認できます。

```
    {
      "method": "Version",
      # important, if method requires no args, enforce "none"
      "args": "none"
      # example method with args, comma seperated
      # "args": "thing1, thing2"
    }
```

コマンドラインの例に我慢できるのであれば、AWS CLI を使用して実行を確認できます。

```
(base) sween @ basenube-pop-os ~/Desktop/BASENUBE
```

```
?? $ &#x25b6; aws lambda invoke --function-name iris-native-lambda --payload '{"metho
d":"Version","args":"none"}' --invocation-type RequestResponse --cli-binary-
format raw-in-base64-out --region us-east-2 --profile default /dev/stdout
{{\"status\":1,\"payload\":\"IRIS for UNIX (Red Hat Enterprise Linux for x86-64) 2020
.2 (Build 210U) Thu Jun 4 2020 15:48:46 EDT\"}"
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

さらに踏み込めば、AWS CLI はエイリアスをサポートしているため、エイリアスを作成して AMS コマンドに完全に統合してみることができます。 以下に CLI エイリアスの例を示します。

```
?? $ &#x25b6; cat ~/.aws/cli/alias
[toplevel]
whoami = sts get-caller-identity

iris =
  !f() {
   aws lambda invoke \
     --function-name iris-native-lambda \
     --payload \
     "{\"method\":\""${1}"\",\"args\":\"none\"}" \
     --invocation-type RequestResponse \
     --log-type None \
     --cli-binary-format raw-in-base64-out \
     gar.json > /dev/null
     cat gar.json
     echo
     echo
  }; f
```

ぜひお試しあれ！

それではお元気で！技術的な議論をお待ちしてます！

#InterSystems IRIS #InterSystems IRIS for Health

ソースURL:https://jp.community.intersystems.com/post/aws-lambda-%E3%81%AE-iris-python-native-api-iris