Statica User Manual

Table of Contents

I. Getting Started

Ruby Quick Start Guide

Installation

When you sign up you will be provided with a unique username and password and a connection string that you can use when configuring your proxy service in your application:

```
http://username:password@brooks.statica.io:9293
```

All requests that you make via this proxy will appear to the destination server to originate from one of the two static IPs you will be assigned when you sign up.

We recommend you store the connection string in an environment variable STATICA_URL. If you have signed up via the add-ons platform on Heroku or cloudControl this variable will be automatically set in your production environment.

Integration

Ruby has many HTTP client libraries but our favourite for HTTP interactions is rest-client. You can run the below example in an irb session and verify that the final IP returned is one of your two static IPs.

```
require "rest-client"
RestClient.proxy = ENV["STATICA_URL"] if ENV["STATICA_URL"]
res = RestClient.get("http://ip.jsontest.com")
puts "Your IP was: #{res.body}"
```

Python Quick Start Guide

Installation

When you sign up you will be provided with a unique username and password and a connection string that you can use when configuring your proxy service in your application:

```
http://username:password@static.statica.io:9293
```

All requests that you make via this proxy will appear to the destination server to originate from one of the two static IPs you will be assigned when you sign up.

We recommend you store the connection string in an environment variable `STATICA_URL`. If you have signed up via the add-ons platform on Heroku or cloudControl this variable will be automatically set in your production environment.

Integration

Python offers many different ways to make HTTP calls but for API requests we recommend the requests library. It allows you to specify an authenticated proxy on a per request basis so you can pick and choose when to route through your static IP address.

```
import requests
import os
proxies = {
"http": os.environ['STATICA_URL'],
"https": os.environ['STATICA_URL']
}
res = requests.get("http://ip.jsontest.com/", proxies=proxies)
print res.text
```

Node .js Quick Start Guide

Installation

When you sign up you will be provided with a unique username and password and a connection string that you can use when configuring your proxy service in your application:

```
http://username:password@static.statica.io:9293
```

All requests that you make via this proxy will appear to the destination server to originate from one of the two static IPs you will be assigned when you sign up.

We recommend you store the connection string in an environment variable STATICA_URL. If you have signed up via the add-ons platform on Heroku or cloudControl this variable will be automatically set in your production environment.

Integration

Statica will work with the standard http library but we recommend using the request library for ease of use.

```
var request = require('request');
var options = {
proxy: process.env.STATICA_URL,
url: 'http://ip.jsontest.com/',
headers: {
'User-Agent': 'node.js'
}
};

function callback(error, response, body) {
if (!error && response.statusCode == 200) {
console.log(body);
}
}
request(options, callback);
```

PHP Quick Start Guide

Installation

When you sign up you will be provided with a unique username and password and a connection string that you can use when configuring your proxy service in your application:

```
http://username:password@static.statica.io:9293
```

All requests that you make via this proxy will appear to the destination server to originate from one of the two static IPs you will be assigned when you sign up.

We recommend you store the connection string in an environment variable `STATICA_URL`. If you have signed up via the add-ons platform on Heroku or cloudControl this variable will be automatically set in your production environment.

Integration

PHP cURL is the easiest way to make HTTP requests via Statica.

```
<?php
function lookup(){
  $statica_env = getenv("STATICA_URL");
  $statica = parse_url($statica_env);

  $proxyUrl = $statica['host'].":".$statica['port'];
  $proxyAuth = $statica['user'].":".$statica['pass'];

  $url = "http://ip.jsontest.com/";

  $ch = curl_init();
  curl_setopt($ch, CURLOPT_URL, $url);
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
  curl_setopt($ch, CURLOPT_PROXY, $proxyUrl);
  curl_setopt($ch, CURLOPT_PROXYAUTH, CURLAUTH_BASIC);
  curl_setopt($ch, CURLOPT_PROXYUSERPWD, $proxyAuth);
  $response = curl_exec($ch);
  return $response;
}

$res = lookup();
print_r($res);

?>
```

.NET Quick Start Guide

Installation

When you sign up you will be provided with a unique username and password and a connection string that you can use when configuring your proxy service in your application:

```
http://username:password@static.statica.io:9293
```

All requests that you make via this proxy will appear to the destination server to originate from one of the two static IPs you will be assigned when you sign up.

We recommend you store the connection string in an environment variable `STATICA_URL`. If you have signed up via the add-ons platform on Heroku or cloudControl this variable will be automatically set in your production environment.

Integration

This example is written in C# and uses the HttpWebRequest class to make a request via Statica.

```
//Extract proxy connection details
string proxyUrl =
Environment.GetEnvironmentVariable("STATICA_URL");
System.Uri proxyUri = new System.Uri(url);
string cleanProxyURL = uri.Scheme + "://" + uri.Host
+":"+uri.Port;
string user = uri.UserInfo.Split(':')[0]
string password = uri.UserInfo.Split(':')[1]
HttpWebRequest request = (HttpWebRequest)
WebRequest.Create("http://ip.jsontest.com/");
WebProxy myProxy = new WebProxy();
Uri newUri = new Uri(cleanProxyURL);
// Associate the newUri object to 'myProxy' object so that new
myProxy settings can be set.
myProxy.Address = newUri;
// Create a NetworkCredential object and associate it with the
// Proxy property of request object.
myProxy.Credentials = new NetworkCredential(user, password);
request.Proxy = myProxy;

HttpWebResponse httpWebResponse = request.GetResponse() as
HttpWebResponse;
// Get the stream containing content returned by the server.
Stream dataStream = httpWebResponse.GetResponseStream();

// Open the stream using a StreamReader for easy access.
StreamReader streamReader = new StreamReader(dataStream);

// Read the content.
String responseFromServer = streamReader.ReadToEnd();

// Write the content.
Console.WriteLine(responseFromServer);

streamReader.Close();
```

```
dataStream.Close();
```

Java Quick Start Guide

Installation

When you sign up you will be provided with a unique username and password and a connection string that you can use when configuring your proxy service in your application:

```
1
http://username:password@static.statica.io:9293
```

All requests that you make via this proxy will appear to the destination server to originate from one of the two static IPs you will be assigned when you sign up.

We recommend you store the connection string in an environment variable `STATICA_URL`. If you have signed up via the add-ons platform on Heroku or cloudControl this variable will be automatically set in your production environment.

Integration

You can use the standard Java libraries with Statica to access HTTP and HTTPS APIs via your static IP. The below examples uses a custom class to encapsulate the Statica proxy logic and an HttpURLConnection to make the HTTP request.

This sample uses the Java 8 Base64 class to encode the authorization String. If using a version less than Java 8 you should use another Base64 encoder implementation, such as the Apache Commons Codec.

```
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.Authenticator;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.zip.GZIPInputStream;
public class HTTPProxyDemo {
  public static void main(String[] args) {
    new HTTPProxyDemo();
  }

  public HTTPProxyDemo(){
    StaticaProxyAuthenticator proxy = new
StaticaProxyAuthenticator();
    String testUrl = "http://ip.jsontest.com/";
    System.out.println(getResponse(proxy, testUrl));
  }

  public String getResponse(StaticaProxyAuthenticator proxy,
String urlToRead) {
        String result = "";
        try {
         URL url = new URL(urlToRead);
           HttpURLConnection conn = (HttpURLConnection)
url.openConnection();
           conn.setRequestProperty("Proxy-Authorization",
"Basic " + proxy.getEncodedAuth());
           conn.setRequestProperty("Accept-Encoding", "gzip");
```

```
            Authenticator.setDefault(proxy.getAuth());
            conn.setRequestMethod("GET");
            InputStream is = conn.getInputStream();
            if(conn.getContentEncoding()!=null &&
conn.getContentEncoding().equalsIgnoreCase("gzip")){
              is = new GZIPInputStream(is);
            }
            byte[] buffer = new byte[1024];
            int len;
            ByteArrayOutputStream bos = new
ByteArrayOutputStream();
            while (-1 != (len = is.read(buffer))) {
              bos.write(buffer, 0, len);
            }
            result = new String(bos.toByteArray());
            is.close();
          } catch (IOException e) {
            e.printStackTrace();
          } catch (Exception e) {
            e.printStackTrace();
          }
          return result;
      }
}
```

```
import java.net.Authenticator;
import java.net.MalformedURLException;
import java.net.PasswordAuthentication;
import java.net.URL;

public class StaticaProxyAuthenticator extends Authenticator{
  private String user, password, host;
  private int port;
  private ProxyAuthenticator auth;

  public StaticaProxyAuthenticator() {
      String proxyUrlEnv = System.getenv("STATICA_URL");
    if(proxyUrlEnv!=null){
      try {
        URL proxyUrl = new URL(proxyUrlEnv);
        String authString = proxyUrl.getUserInfo();
        user = authString.split(":")[0];
        password = authString.split(":")[1];
        host = proxyUrl.getHost();
        port = proxyUrl.getPort();
        auth = new ProxyAuthenticator(user,password);
        setProxy();
      } catch (MalformedURLException e) {
        e.printStackTrace();
      }
    }
    else{
      System.err.println("You need to set the environment
variable STATICA_URL!");
    }

    }
```

```
    private void setProxy(){
      System.setProperty("http.proxyHost", host);
        System.setProperty("http.proxyPort",
String.valueOf(port));
        System.setProperty("https.proxyHost",host);
        System.setProperty("https.proxyPort",
String.valueOf(port));
    }

    public String getEncodedAuth(){
      //If not using Java8 you will have to use another Base64
encoded, e.g. apache commons codec.
      String encoded =
java.util.Base64.getEncoder().encodeToString((user + ":" +
password).getBytes());
      return encoded;
    }

    public ProxyAuthenticator getAuth(){
      return auth;
    }

    class ProxyAuthenticator extends Authenticator {

        private String user, password;

        public ProxyAuthenticator(String user, String password)
{
            this.user = user;
            this.password = password;
        }

        protected PasswordAuthentication
getPasswordAuthentication() {
            return new PasswordAuthentication(user,
password.toCharArray());
        }
    }

}
```

SOCKS Quick Start Guide

Statica provides a wrapper script that transparently forwards all outbound TCP traffic through your Static IP. This is language independent but there are known issues with certain Node.js connections hanging so please contact us if you have any issues.

If you're not sure whether to use the SOCKS proxy check out our HTTP vs SOCKS article.

Please note: The wrapper is not compatible with OSX or Windows. We recommend using a Virtual Machine running Ubuntu for development testing if your main development environment does not run Linux.

Installing the Statica wrapper

Download and extract the wrapper in your app directory

```
$ cd /home/myuser/my-app

$ curl https://s3.amazonaws.com/statica-releases/statica-
socksify-latest.tar.gz | tar xz

$ echo "STATICA-LICENSE.txt" >> .gitignore

$ git add bin/statica vendor/dante

$ git commit -m "Add Statica socksify"
```

Now you can prepend the *statica* wrapper to your standard commands to transparently route all traffic through your Static IPs. For example to run a Rails server:

```
bin/statica rails s
```

Controlling what traffic goes through proxy

You can provide a standard subnet mask to only route traffic to certain IP subnets via the STATICA_MASK environment variable. The mask supports sub-network specifications (e.g., 192.0.2.22/24), single addresses (192.0.2.22/32), host names (e.g., ftp.example.org), or domain names (e.g., .example.org). A domain name specification will match any host in that domain.

```
export STATICA_MASK="100.30.68.0/24"
```

All outbound traffic to 100.30.68.* would be routed via your Static IPs. Multiple masks can be provided by comma separating the mask values:

```
export STATICA_MASK="100.30.68.0/24,99.29.68.0/24"
```

Choosing between the HTTP and SOCKS proxies

With every Statica plan you have the option of using our HTTP or SOCKS5 proxies. This article outlines the differences and what factors should influence your decision.

HTTP vs SOCKS comparison

The majority of our customers use our HTTP proxy. This allows you to route any HTTP calls via our proxy including secure requests over HTTPS. HTTP proxies are natively supported in most of the common programming languages and HTTP client libraries so are easy to integrate.

SOCKS proxies are more versatile as they operate at a lower level than HTTP and can proxy TCP connections to arbitrary IP addresses. This allows you to proxy higher level protocol interactions like FTP or LDAP. SOCKS is supported at the socket level in a lot of the major languages but most client libraries do not natively support it which makes it harder to integrate in to your application.

Due to the ease of integration if you are accessing an HTTP or HTTPS API you should use our HTTP proxy. If you are using a different protocol then you should switch to SOCKS.

Common HTTP Use Cases

- Accessing an HTTP API
- Accessing an HTTPS API

Common SOCKS Use Cases

- Accessing an MySQL database
- Accessing an LDAP service
- Transferring files via Secure FTP

II. Using with Node

Getting started with Node.js standard http library

This example assumes your STATICA_URL environment variable is set and contains your unique connection string.

To access an HTTP API you can use the standard HTTP library in Node.js but must ensure you correctly set the “Host” header to your target hostname, not the proxy hostname.

```
var http, options, proxy, url;
http = require("http");

url = require("url");

proxy = url.parse(process.env.STATICA_URL);
target = url.parse("http://ip.jsontest.com/");

options = {
hostname: proxy.hostname,
port: proxy.port || 80,
path: target.href,
headers: {
"Proxy-Authorization": "Basic " + (new
Buffer(proxy.auth).toString("base64")),
"Host" : target.hostname
}
};

http.get(options, function(res) {
res.pipe(process.stdout);
return console.log("status code", res.statusCode);
});
```

Accessing a MySQL Database via a Static IP from Node.js

You can route all database traffic via a Static IP in Node.js using Statica. Currently there is a limitation in that you can't use the built in connection pooling so this is not recommended for high traffic applications.

```
var mysql = require('mysql2');
var url = require("url");
var SocksConnection = require('socksjs');
var remote_options = {
host:'your-database.eu-west-1.rds.amazonaws.com',
port: 3306
};
var proxy = url.parse(process.env.STATICA_URL);
var auth = proxy.auth;
var username = auth.split(":")[0]
var pass = auth.split(":")[1]

var sock_options = {
host: proxy.hostname,
port: 1080,
user: username,
pass: pass
}
var sockConn = new SocksConnection(remote_options,
sock_options)
var dbConnection = mysql.createConnection({
user: 'dbuser',
database: 'dbname',
password: 'dbpassword',
stream: sockConn
});
dbConnection.query('SELECT 1+1 as test1;', function(err, rows,
fields) {
if (err) throw err;

console.log('Result: ', rows);
sockConn.dispose();
});
dbConnection.end();
```

Accessing a SOAP service from Node.js with a Static IP

The node-soap library is a great library for connecting to SOAP services from your node.js app but it does not support sending traffic over authenticated proxies.

We have forked node-soap to read in the QUOTAGUARDSTATIC_URL environment variable and route all traffic via our proxy so that you can send requests via your Static IP. Statica uses should manually set the QUOTAGUARDSTATIC_URL variable to match their STATICA_URL variable for this to work.

To use replace your current node-soap dependency with our repo https://github.com/quotaguard/node-soap

For example in your package.json:

```
{
"name": "foo",
"version": "0.0.0",
"dependencies": {
"node-soap": "quotaguard/node-soap"
}
}
```

III. Using with PHP

Accessing an FTP server from a Static IP in PHP

You can access an FTP server via Statica from your PHP application so that your traffic always appears to come from the same IP address. This solution uses PHP cURL and our SOCKS5 proxy which is accessible on port 1080:

Using with an unauthenticated FTP Server

```
<?php
$statica_env = getenv("STATICA_URL");
$statica = parse_url($statica_env);
$proxyUrl = $statica['host'].":1080";
$proxyAuth = $statica['user'].":".$statica['pass'];
print $proxyUrl;
$curl = curl_init();
curl_setopt($curl, CURLOPT_URL,"ftp://ftp.gnu.org");
curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($curl, CURLOPT_PROXY, $proxyUrl);
curl_setopt($curl, CURLOPT_PROXYAUTH, CURLAUTH_BASIC);
curl_setopt($curl, CURLOPT_PROXYUSERPWD, $proxyAuth);
curl_setopt($curl, CURLOPT_PROXYTYPE, CURLPROXY_SOCKS5);

$result = curl_exec ($curl);
curl_close ($curl);
print $result;
?>
```

Using with an authenticated FTP Server

If your FTP server requires a username and password you just need to set the extra CURLOPT_USERPWD parameter.

```
<?php
$username = "myftpuser";
$password = "myftppw";
$statica_env = getenv("STATICA_URL");
$statica = parse_url($statica_env);
$proxyUrl = $statica['host'].":1080";
$proxyAuth = $statica['user'].":".$statica['pass'];
print $proxyUrl;
$curl = curl_init();
// This example will not work with this FTP server as it does
not require username/password!
curl_setopt($curl, CURLOPT_URL,"ftp://ftp.gnu.org");
curl_setopt($ch, CURLOPT_USERPWD, "$username:$password");
curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($curl, CURLOPT_PROXY, $proxyUrl);
curl_setopt($curl, CURLOPT_PROXYAUTH, CURLAUTH_BASIC);
curl_setopt($curl, CURLOPT_PROXYUSERPWD, $proxyAuth);
curl_setopt($curl, CURLOPT_PROXYTYPE, CURLPROXY_SOCKS5);

$result = curl_exec ($curl);
curl_close ($curl);
print $result;
?>
```

IV. Platform Integrations

Extract Statia credentials from VCAP_SERVICES

Working with VCAP_SERVICES

Our documentation assumes that your Statica credentials are available in the form of a connection string in the STATICA_URL environment variable. On CloudFoundry based platforms like IBM Bluemix, Pivotal and RedHat Openshift this is not the case so you need slightly different code to access your Statica credentials by extracting them from the VCAP_SERVICES environment variable.

The VCAP_SERVICES entry looks like the following:

```
{"statica":[{"name":"StaticaInstance","label":"statica","tags":
[],"plan":"starter","credentials":
{"STATICA_URL":STATICA_URL}}]}
```

Extract the variable using the snippets below and then continue with our standard documented solutions.

In Java

```
String vcapServices = System.getenv("VCAP_SERVICES");
JsonRootNode root = new JdomParser().parse(vcapServices);
JsonNode mysqlNode = root.getNode("statica");
JsonNode credentials =
mysqlNode.getNode(0).getNode("credentials");
String staticaURL = credentials.getStringValue("STATICA_URL");
```

In Ruby

```
credentials = proxyURL = ''
if !ENV['VCAP_SERVICES'].blank?
  JSON.parse(ENV['VCAP_SERVICES']).each do |k,v|
    if !k.scan("statica").blank?
      credentials = v.first.select {|k1,v1| k1 ==
"credentials"}["credentials"]
      proxyURL = credentials["STATICA_URL"]
    end
  end
end
```

In Python

```
statica_service = json.loads(os.environ['VCAP_SERVICES'])
['statica'][0]
credentials = statica_service['credentials']
statica_url = credentials['STATICA_URL']
```

In Node.js/Javascript

```
var vcap_services = JSON.parse(process.env.VCAP_SERVICES)
var staticaUrl = vcap_services['statica']
[0].credentials.STATICA_URL
```

In PHP

```
$vcap = json_decode(getenv("VCAP_SERVICES"),true);
$statica_env = $vcap["statica"][0]["credentials"]
["STATICA_URL"];
```

IBM Bluemix

Statica is available on the IBM Bluemix Catalog. Binding the add-on will provide you with access to our HTTP and SOCKS proxies with your connection string automatically populated in the VCAP_SERVICES environment variable.

Getting Started

For examples on how to integrate Statica with your application check out our Quick Start Guides below. Before you head off there be aware of the one major Bluemix difference to our standard documentation, working with VCAP_SERVICES.

Working with VCAP_SERVICES

Learn how to extract your Statica credentials by following our VCAP_SERVICES guide.

With the credentials you can then follow one of our quick start guides to get up and running.

Generic Quick Start Guides

- Java
- Ruby
- Python
- Node.js
- PHP

Rails Starter App

If you're developing in Rails you can clone our IBM Bluemix Rails with Statica sample app and get going in minutes.

V. Using with Ruby

Getting started with the rest-client gem

Integrating with the rest-client gem is a one-liner. This will print out one of your Statica IPs:

```
require "rest-client"

RestClient.proxy = ENV["STATICA_URL"] if ENV["STATICA_URL"]

res = RestClient.get("http://ip.jsontest.com")

puts "Your IP was: #{res.body}"
```

Getting started with the httparty gem

HTTParty is a Ruby library for making HTTP requests and integrates easily with Statica:

The following code snippet will print out one of your Static IP addresses.

```
require 'httparty'

proxy = URI(ENV["STATICA_URL"])

options = {http_proxyaddr:
proxy.host,http_proxyport:proxy.port,
http_proxyuser:proxy.user, http_proxypass:proxy.password}
response = HTTParty.get('http://ip.jsontest.com/',options)

puts response.body, response.code, response.message,
response.headers.inspect
```

Getting started with the httpclient gem

httpclient is a Ruby gem allowing you to execute HTTP commands. You can route these commands easily via a Static IP by integrating with Statica:

```
require 'httpclient'

statica = URI(ENV["STATICA_URL"])
proxy_url = statica
client = HTTPClient.new(proxy_url)
prx_user = statica.user
prx_password = statica.password
client.set_proxy_auth(prx_user, prx_password)
res = client.get("http://ip.jsontest.com/")
puts res.body
```

Sending files via a Static IP with Ruby SFTP

Secure FTP allows you to transfer data securely between servers. You can use the Ruby SFTP library to route SFTP transfers via Statica allowing you to lock down access to our Static IP addresses:

```
require 'net/ssh'
require 'uri'
require 'net/sftp'
require 'net/ssh/proxy/http'
statica = URI(ENV["STATICA_URL"])

proxy =
Net::SSH::Proxy::HTTP.new(statica.host,statica.port, :user =>
statica.user,:password=>statica.password)

Net::SSH.start('1.1.1.1','sftpuser',
{:port => 22,
:proxy => proxy,
:password => 'sftppw'}) do |ssh|

ssh.sftp.connect do |sftp|
sftp.dir.foreach("/") do |entry|
puts entry.longname
end
end
end
```

Access RETS services via a Static IP with the ruby-rets gem

ruby-rets is a Ruby client to allow you retrieve real estate data from RETS servers. You can easily integrate it with Statica for IP locked RETS servers:

```
require "ruby-rets"
proxy = URI(ENV["STATICA_URL"])

client = RETS::Client.login(:url => "http://rets.example.com:
6103/rets2_1/Login", :username => "<retsuser>",
:password => "<rets-password>", :http => {
:proxy => {
:address => proxy.host,
:port => proxy.port,
:username => proxy.user,
:password => proxy.password
}
})
p client
```

Getting started with the Net::HTTP

Net::HTTP is Ruby's native HTTP library. For any complex web service interactions we recommend rest-client but for simple cases net/http is ok.

```
require 'uri'
require 'net/http'

proxy = URI(ENV["STATICA_URL"])
h = Net::HTTP.new('ip.jsontest.com', 80,proxy.host, proxy.port,
proxy.user, proxy.password)
response = h.get("/")
p response.body
```

Accessing a firewalled LDAP Server from Ruby via Statica

As part of our service we offer a SOCKS5 proxy which is accessible on your assigned Statica server on port 1080. SOCKS5 provides TCP level proxying so can handle all sorts of traffic including LDAP.

Ruby has a great LDAP library which can handle the bulk of the work but at the moment it doesn't support sending traffic via a proxy. To get around this we have produced a patched version of net-ldap which routes all LDAP traffic through our SOCKS proxy. This requires a patched version of socksify-ruby so you just need to install these two gems from our Github repository.

```
#Gemfile
gem 'socksify', github: "quotaguard/socksify-ruby", branch:
"master"
gem 'net-ldap', github: "quotaguard/ruby-net-ldap", branch:
"master"
```

Once installed set a QUOTAGUARDSTATIC_URL variable to match your STATICA_URL variable. The patched Ruby Net LDAP client looks for a hardcoded QUOTAGUARDSTATIC_URL variable (the previous name for our Statica service).

VI. Using with Phyton

Getting started with urllib2

This example assumes your STATICA_URL environment variable is set and contains your unique connection string.
urllib2 is a basic library used for HTTP communication in Python and uses environment variables to set a proxy service.

In your application initialization you should set the http_proxy variable to match the STATICA_URL.

Assign Statica to your environment's http_proxy variable

```
os.environ['http_proxy'] = os.environ['STATICA_URL']
```

To test in the Python interpreter

```
import urllib2, os
os.environ['http_proxy'] = os.environ['STATICA_URL']
url = 'http://ip.jsontest.com/'
proxy = urllib2.ProxyHandler()
opener = urllib2.build_opener(proxy)
in_ = opener.open(url)
res = in_.read()
print res
```

VII. Using with .NET

Accessing a SOAP service from .NET MVC Controller with a Static IP

This example assumes your STATICA_URL environment variable is set and contains your unique connection string.

Here we show how to use Statica to access a SOAP service from a .NET MVC web controller. This example should also apply to other WebRequests made from an MVC controller.

A common use case is from your MVC controller you want to access an IP-whitelisted service that uses SOAP but the web service object does not have a "proxy" property that allows you to specify the Statica proxy service.

Solution: Create a proxy object and I use it as a property of the WebRequest MVC class, which will send all following requests via the proxy.

The solution is split in to two parts, the configuration in your web.config (which on Azure can be done from your management console) and the MVC Controller itself.

MVC Controller

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Bank.Webservices; // fictitious namespace name for your
webservice's reference
using System.Diagnostics;
using System.Xml;
using System.Net;
using System.Web.Configuration;

public class PaymentController : Controller
{
[HttpPost]
// this is usually a form result, so attribute of action is
HttpPost
public ActionResult Index(string anyTokenYouUseWithYourBank)
{
string StaticaProxyAddress =
WebConfigurationManager.AppSettings["Statica_Proxy_URI"];
// to get the Proxy Uri from Web.config

string StaticaUser =
WebConfigurationManager.AppSettings["Statica_User"];
// to get the Proxy User from Web.config

string StaticaPassword =
WebConfigurationManager.AppSettings["Statica_Password"];
// to get the Proxy Password from Web.config
```

```
var proxy = new WebProxy(StaticaProxyAddress);
// we create a new proxy object

proxy.Credentials = new NetworkCredential(StaticaUser,
StaticaPassword);
// we give the proxy the Statica credentials

WebRequest.DefaultWebProxy = proxy;
// thanks to WebRequest, from here on every request of this
controller will pass through the proxy

var soapClientObject = new WSBankSoapClient(); //fictitious
name for the banks's web service

var result =
soapClientObject .GrabSomethingOnTheNet(anyTokenYouUseWithYourB
ank);
// now our WS call goes passes Proxy and in the variable result
we have the web service's result

// we use the result here

return Redirect("http://newpage");

}
}
```

Configuration (web.config)

The section to add the keys to is <configuration><appSettings> (which you can also directly modify in the Azure Console):

```
<appSettings>
...
<!-- keys for Statica-->
<add key="Statica_Proxy_URI" value="http://
yourStaticaUser:yourStaticaPassword@eu-west-1-
babbage.statica.io:9293"/>
<add key="Statica_User" value="yourStaticaUser"/>
<add key="Statica_Password" value="yourStaticaPassword"/>
<!-- end of keys for Statica-->
...
</appSettings>
```

This solution was kindly contributed by Riccardo Moschetti, a Microsoft MCSD Web Applications specialist.