

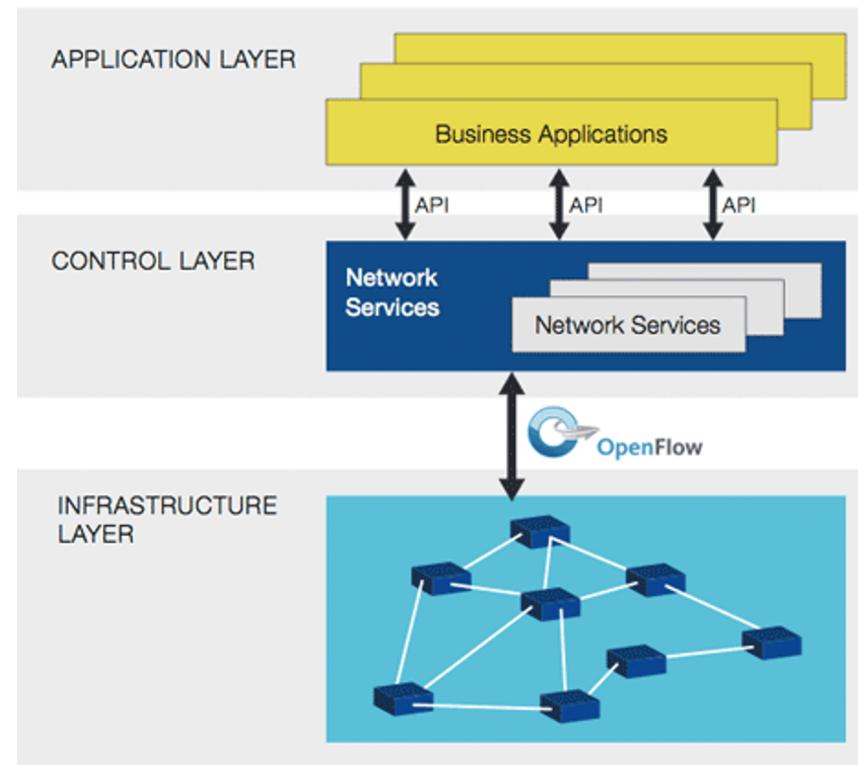
Complex networks

Floodlight RESTful interface

Antonio Virdis
Assistant Professor@ University of Pisa
antonio.virdis@unipi.it

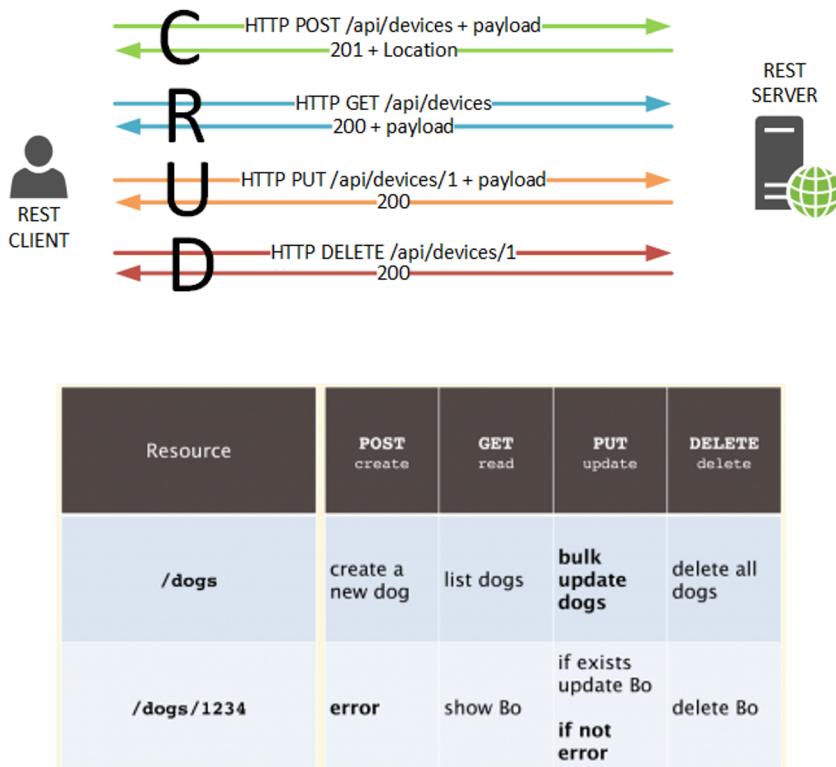
SDN Northbound interface

- Core advantage of Software Defined Networks over traditional deployments is their re-programmability
- Such changes are usually applied automatically by external applications and monitoring process to react to events (e.g. variations in the load)
- To this aim, a northbound interface is exposed by controllers

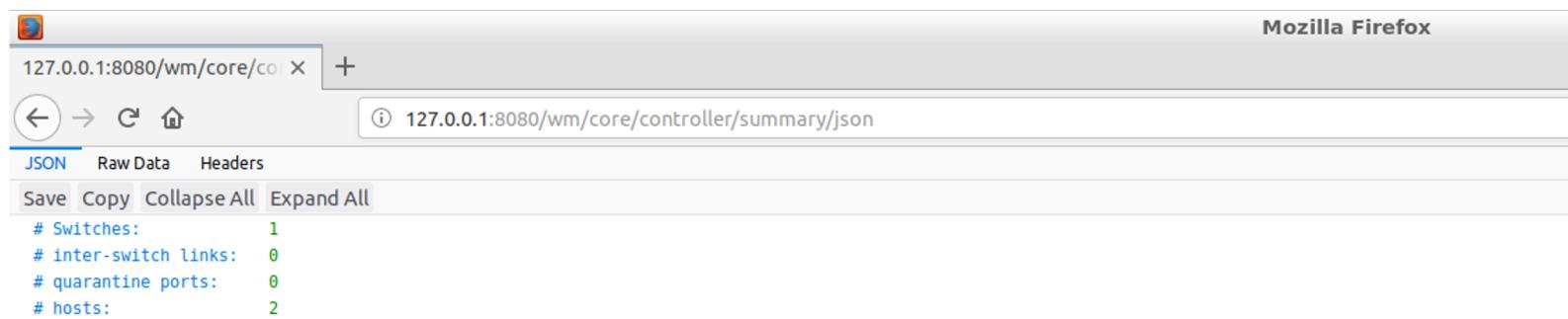


RESTful APIs

- Floodlight defines the APIs of the northbound interface using the RESTful paradigm
- Each controller exposes a set of resources that can be used by applications to retrieve network information or issue configuration changes or trigger operations



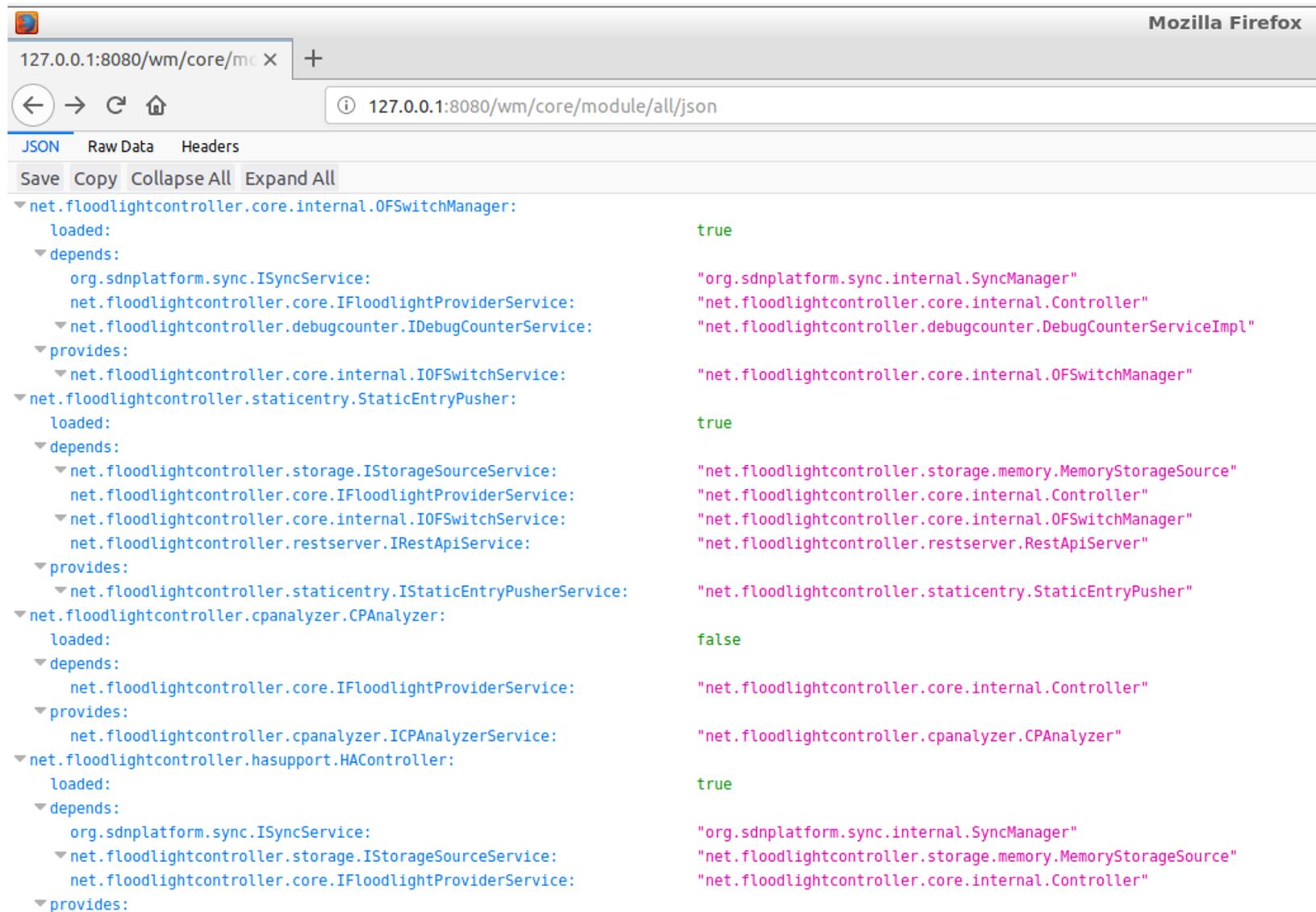
http://192.168.57.____:8080/wm/core/controller/summary/json



The screenshot shows a Mozilla Firefox browser window. The address bar displays "127.0.0.1:8080/wm/core/controller/summary/json". The main content area shows a JSON response with the following data:

```
# Switches:      1
# inter-switch links: 0
# quarantine ports: 0
# hosts:         2
```

[http:// 192.168.57.1:8080/wm/core/module/all/json](http://192.168.57.1:8080/wm/core/module/all/json)



The screenshot shows a Mozilla Firefox browser window displaying a JSON object. The URL in the address bar is 127.0.0.1:8080/wm/core/module/all/json. The JSON structure is as follows:

```
127.0.0.1:8080/wm/core/mo + Mozilla Firefox  
127.0.0.1:8080/wm/core/module/all/json  
JSON Raw Data Headers  
Save Copy Collapse All Expand All  
▼ net.floodlightcontroller.core.internal.OFSwitchManager:  
  loaded: true  
  depends:  
    org.sdnplatform.sync.ISyncService: "org.sdnplatform.sync.internal.SyncManager"  
    net.floodlightcontroller.core.IFloodlightProviderService: "net.floodlightcontroller.core.internal.Controller"  
    ▼ net.floodlightcontroller.debugcounter.IDebugCounterService: "net.floodlightcontroller.debugcounter.DebugCounterServiceImpl"  
  provides:  
    ▼ net.floodlightcontroller.core.internal.IOFSwitchService: "net.floodlightcontroller.core.internal.OFSwitchManager"  
▼ net.floodlightcontroller.staticentry.StaticEntryPusher:  
  loaded: true  
  depends:  
    ▼ net.floodlightcontroller.storage.IStorageSourceService: "net.floodlightcontroller.storage.memory.MemoryStorageSource"  
      net.floodlightcontroller.core.IFloodlightProviderService: "net.floodlightcontroller.core.internal.Controller"  
    ▼ net.floodlightcontroller.core.internal.IOFSwitchService: "net.floodlightcontroller.core.internal.OFSwitchManager"  
      net.floodlightcontroller.restserver.IRest ApiService: "net.floodlightcontroller.restserver.RestApiServer"  
  provides:  
    ▼ net.floodlightcontroller.staticentry.IStaticEntryPusherService: "net.floodlightcontroller.staticentry.StaticEntryPusher"  
▼ net.floodlightcontroller.cpanalyzer.CPAnalyzer:  
  loaded: false  
  depends:  
    net.floodlightcontroller.core.IFloodlightProviderService: "net.floodlightcontroller.core.internal.Controller"  
  provides:  
    net.floodlightcontroller.cpanalyzer.ICPAnalyzerService: "net.floodlightcontroller.cpanalyzer.CPAnalyzer"  
▼ net.floodlightcontroller.hasupport.HAController:  
  loaded: true  
  depends:  
    org.sdnplatform.sync.ISyncService: "org.sdnplatform.sync.internal.SyncManager"  
    ▼ net.floodlightcontroller.storage.IStorageSourceService: "net.floodlightcontroller.storage.memory.MemoryStorageSource"  
      net.floodlightcontroller.core.IFloodlightProviderService: "net.floodlightcontroller.core.internal.Controller"  
  provides:
```

Configuring SDN firewall

Enable/disable firewall

Resource	Command	body
/wm/firewall/module/enable/json	PUT	
/wm/firewall/module/disable/json	PUT	

Add firewall rule

Resource	Command	body
/wm/firewall/rules/json	POST	{"src-ip": "10.0.0.2/32", "dst-ip": "10.0.0.1/32", "dl-type": "ARP"}
/wm/firewall/rules/json	POST	{"src-ip": "10.0.0.2/32", "dst-ip": "10.0.0.1/32", "nwproto": "UDP"}

Test 1

- Enable firewall and check ping
- Add rules and check ping ✗
- Check udp connectivity (e.g.; through nc) ✗
- Check connectivity on the reverse-path ✗

Expose REST resources

On a high-level, there are four steps to implementing a REST API in Floodlight:

1. Get the `IRest ApiService`
2. Define the URIs we want for our module
3. (Implement the URIs defined in our module)
4. Tell Floodlight our new REST API exists

Floodlight RESTful API – STEP 1

- Each Floodlight module can expose a RESTful interface for external applications. REST interface for the *loadbalancer*.
- First step, add the IRest ApiService among the dependences and retrieve a reference to it

```
protected IRest ApiService rest ApiService; // Reference to the Rest API service

...
@Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
    Collection<Class<? extends IFloodlightService>> l = new ArrayList<Class<?
        extends IFloodlightService>>();
    ...
    // Add among the dependences the RestApi service
    l.add(IRest ApiService.class);
    return l;
}
...
@Override
public void init(FloodlightModuleContext context) throws FloodlightModuleException {
    ...
    // Retrieve a pointer to the rest api service
    rest ApiService = context.getServiceImpl(IRest ApiService.class);
}
```

Floodlight RESTful API – STEP 2

- Second step, create a class representing the RESTful interface

```
public class LoadBalancerWebRoutable implements RestletRoutable {  
    /**  
     * Create the Restlet router and bind to the proper resources.  
     */  
    @Override  
    public Restlet getRestlet(Context context) {  
        Router router = new Router(context);  
  
        // Add some pre-defined REST resources available in the floodlight framework  
  
        // This resource will show some summary stats on the controller  
        router.attach("/controller/summary/json", ControllerSummaryResource.class);  
  
        // This resource will show the list of modules loaded in the controller  
        router.attach("/module/loaded/json", LoadedModuleLoaderResource.class);  
  
        // This resource will show the list of switches connected to the controller  
        router.attach("/controller/switches/json", ControllerSwitchesResource.class);  
  
        return router;  
    }  
    @Override  
    public String basePath() {  
        // The root path for the resources  
        return "/lb";  
    }  
}
```

Floodlight RESTful API – STEP 4

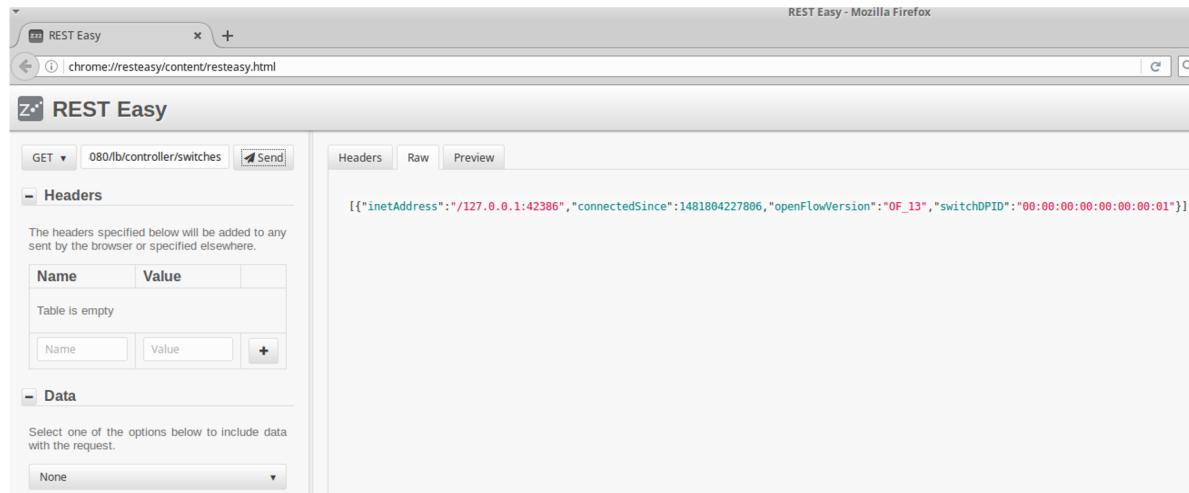
- Third step, add the REST interface at module's startup

```
//within LoadBalancer class

@Override
public void startUp(FloodlightModuleContext context) throws FloodlightModuleException
{
    ...
    // Add as REST interface
    rest ApiService.addRestletRoutable(new LoadBalancerWebRoutable());
}
```

Floodlight RESTful API – TEST

- The module is now ready to expose some pre-defined floodlight resources, e.g. to show general statistics on the SDN network
- The REST interface can be tested through one of the REST add-ons for browsers, for example “RESTED” for Firefox or “ARC” for Chrome
- The REST interface is exposed by floodlight on the port 8080
- Test the resource
<http://192.168.57.1:8080/lb/controller/switches/json> to get a list of the switches connected



Floodlight RESTful API – Custom res (STEP 3)

On a high-level, there are four steps to implementing a REST API in Floodlight:

1. Get the `IRest ApiService`
2. Define the URIs we want for our module
3. (Implement the URIs defined in our module)
4. Tell Floodlight our new REST API exists

Floodlight RESTful API – Custom resources (STEP 3)

- In order to add custom resources to your module, an additional class must be defined
- New resources can be defined only through external classes as this one for example that simply returns a custom JSON message:

```
//...
import org.restlet.resource.Get;
//...

public class TestResource extends ServerResource {

    @Get("json")
    public Map<String, Object> Test() {

        Map<String, Object> info = new HashMap<String, Object>();

        info.put("name", "value");

        return info;
    }
}
```

Floodlight RESTful API – Custom resources

- As external class, such custom resources can not access to internal module information, neither trigger internal operations
- In order to export functionalities and data to other classes the module must define and implement a public interface available to other module as follow:

```
// Service interface for the module
// This interface will be used to interact with other modules
// Export here all the methods of the class that are likely used by other modules

public interface ILoadBalancerREST extends IFloodlightService
{
    // Method exposed by the module
    // Method to retrieve the info on the real servers of the pool
    public Map<String, Object> getServersInfo();

    // Method to set the hardtimeout for load balancing
    public void setHardTimeout(int newValue);
}
```

Floodlight RESTful API – Custom resources

- The module must include the interface through the following steps:
 - Add the implementation of the interface:

```
public class LoadBalancerREST implements  
    IOFMessageListener, IFloodlightModule, ILoadBalancerREST {
```

```
@Override  
public Collection<Class<? extends IFloodlightService>> getModuleServices() {  
    Collection<Class<? extends IFloodlightService>> l =  
        new ArrayList<Class<? extends IFloodlightService>>();  
    l.add(ILoadBalancerREST.class);  
    return l;  
}  
@Override  
public Map<Class<? extends IFloodlightService>, IFloodlightService> getServiceImpls()  
{  
    Map<Class<? extends IFloodlightService>, IFloodlightService> m =  
        new HashMap<Class<? extends IFloodlightService>, IFloodlightService>();  
    m.put(ILoadBalancerREST.class, this);  
    return m;  
}
```

Floodlight RESTful API – Custom resources

- The module must implement the methods of the interface:

```
@Override  
public Map<String, Object> getServersInfo(){  
    Map<String, Object> info = new HashMap<String, Object>();  
    for( int i = 0; i < SERVERS_MAC.length; i++)  
    {  
        info.put( SERVERS_IP[i] , SERVERS_MAC[i]);  
    }  
    return info;  
}  
  
@Override  
public void setHardTimeout(int newValue){  
    HARD_TIMEOUT = (short) newValue;  
    System.out.println("Timeout value changed to " + newValue + "\n");  
}
```

Floodlight RESTful API – Custom resources

- For each REST resource its implementation can invoke the methods of the public interface, e.g. a resource to retrieve the list of the servers in the pool

```
// Implementation of a resource to retrieve the list of the servers in the pool
public class GetServerInfo extends ServerResource {
    @Get("json")
    public Map<String, Object> Test() {
        ILoadBalancerREST lb =
            (ILoadBalancerREST)getContext().getAttributes()
                .get(ILoadBalancerREST.class.getCanonicalName());
        return lb.getServerInfo();
    }
}
```

Floodlight RESTful API – Custom resources

- Resource to set the value for load balancing

```
public class ChangePeriod extends ServerResource {  
    @Post  
    public String store(String fmJson) {  
  
        // Check if the payload is provided  
        if(fmJson == null){  
            return new String("No attributes");  
        }  
  
        // Parse the JSON input  
        ObjectMapper mapper = new ObjectMapper();  
        try {  
            JsonNode root = mapper.readTree(fmJson);  
            // Get the field hardtimeout  
            int newValue = Integer.parseInt(root.get("hardtimeout").asText());  
  
            ILoadBalancerREST lb = (ILoadBalancerREST)  
                getContext().getAttributes()  
                .get(ILoadBalancerREST.class.getCanonicalName());  
  
            lb.setHardTimeout(newValue);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return new String("OK");  
    }  
}
```


Floodlight RESTful API – TEST

- Test the resource `http://192.168.57.XXX:8080/lb/timeout/json`
- The argument must be specified in JSON format in the POST payload as follow:

The screenshot shows the REST Easy interface. On the left, the request details are set: method is POST, URL is `http://127.0.0.1:8080/lb/timeout/json`, and the Headers tab is selected. In the Data section, the Content-Type is set to `application/json`, and the payload is `{"hardtimeout": "30"}`. On the right, the response is displayed under the Headers tab, showing a 200 OK status. The response headers are listed in a table:

Name	Value
Content-Length	2
Content-Type	application/json; charset=UTF-8
Date	Sat, 17 Dec 2016 11:36:03 GMT
Accept-Ranges	bytes
Server	Restlet-Framework/2.3.1
Vary	Accept-Charset, Accept-Encoding, Acc
Access-Control-Expose-Headers	Authorization, Link
Access-Control-Allow-Credentials	true
Access-Control-Allow-Methods	GET,POST,PUT,DELETE
Access-Control-Allow-Origin	*
Access-Control-Allow-Headers	*
Connection	keep-alive

Test 2

- Modify the LoadBalancer implementation in order to introduce a REST interface to retrieve the list of servers and set the hardtimeout

References

- *Floodlight REST tutorial:*
<https://floodlight.atlassian.net/wiki/display/floodlight/controller/How+to+add+a+REST+API+to+a+Module>
- RESTED Firefox extension:
<https://addons.mozilla.org/it/firefox/addon/rested/>