CLOUD + IOT

UNIT – IV

Autobahn

The **Autobahn** project provides open-source implementations of the The WebSocket Protocol and The Web Application Messaging Protocol (WAMP) network protocols.

Conti…

WebSocket allows **bidirectional real-time messaging** on the Web and WAMP adds asynchronous **Remote Procedure Calls** and **Publish & Subscribe** on top of WebSocket.

WAMP is **ideal for distributed, multi-client and server applications**, such as multi-user database-driven business applications, sensor networks (IoT), instant messaging or MMOGs (massively multi-player online games).

Key WAMP concepts

Transport: Transport is channel that connects two peers. The default transport for WAMP is WebSocket. WAMP can run over other transports as well which support message-based reliable bi-directional communication.

Session: Session is a conversation between two peers that runs over a transport.

Client: Clients are peers that can have one or more roles. In publish-subscribe model client can have following roles:

- **Publisher:** Publisher publishes events (including payload) to the topic maintained by the Broker.
- **Subscriber:** Subscriber subscribes to the topics and receives the events including the payload.

In RPC model client can have following roles:

- **Caller:** Caller issues calls to the remote procedures along with call arguments.
- **Callee:** Callee executes the procedures to which the calls are issued by the caller and returns the results back to the caller.

Conti…

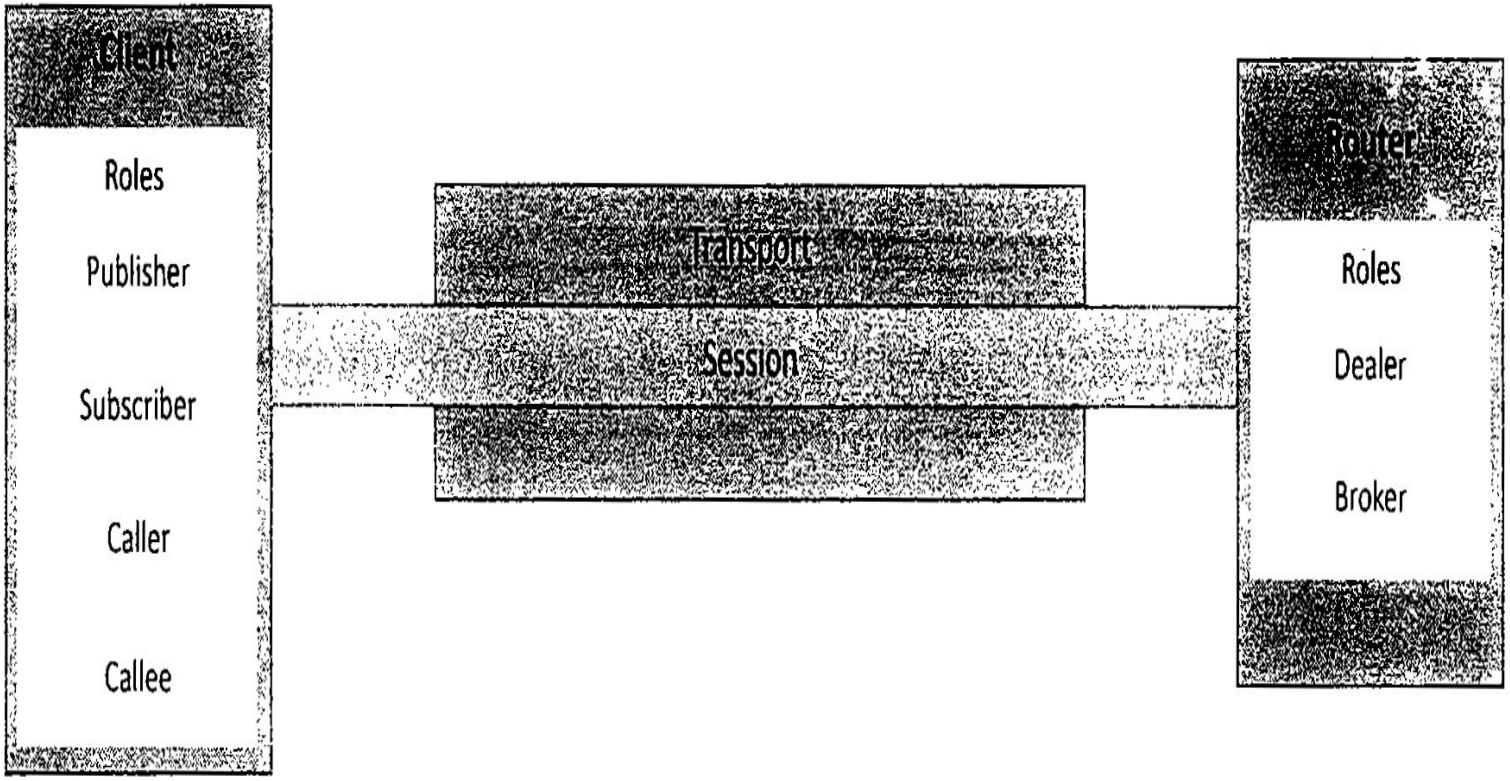
- **Router:** Routers are peers that perform generic call and event routing. In publish-subscribe model Router has the role of a Broker:
  - **Broker:** Broker acts as a router and routes messages published to a topic to all subscribers subscribed to the topic.

    In RPC model Router has the role of a Broker:
  - **Dealer:** Dealer acts a router and routes RPC calls from the Caller to the Callee and routes results from Callee to Caller.
- **Application Code:** Application code runs on the Clients (Publisher, Subscriber, Callee or Caller).

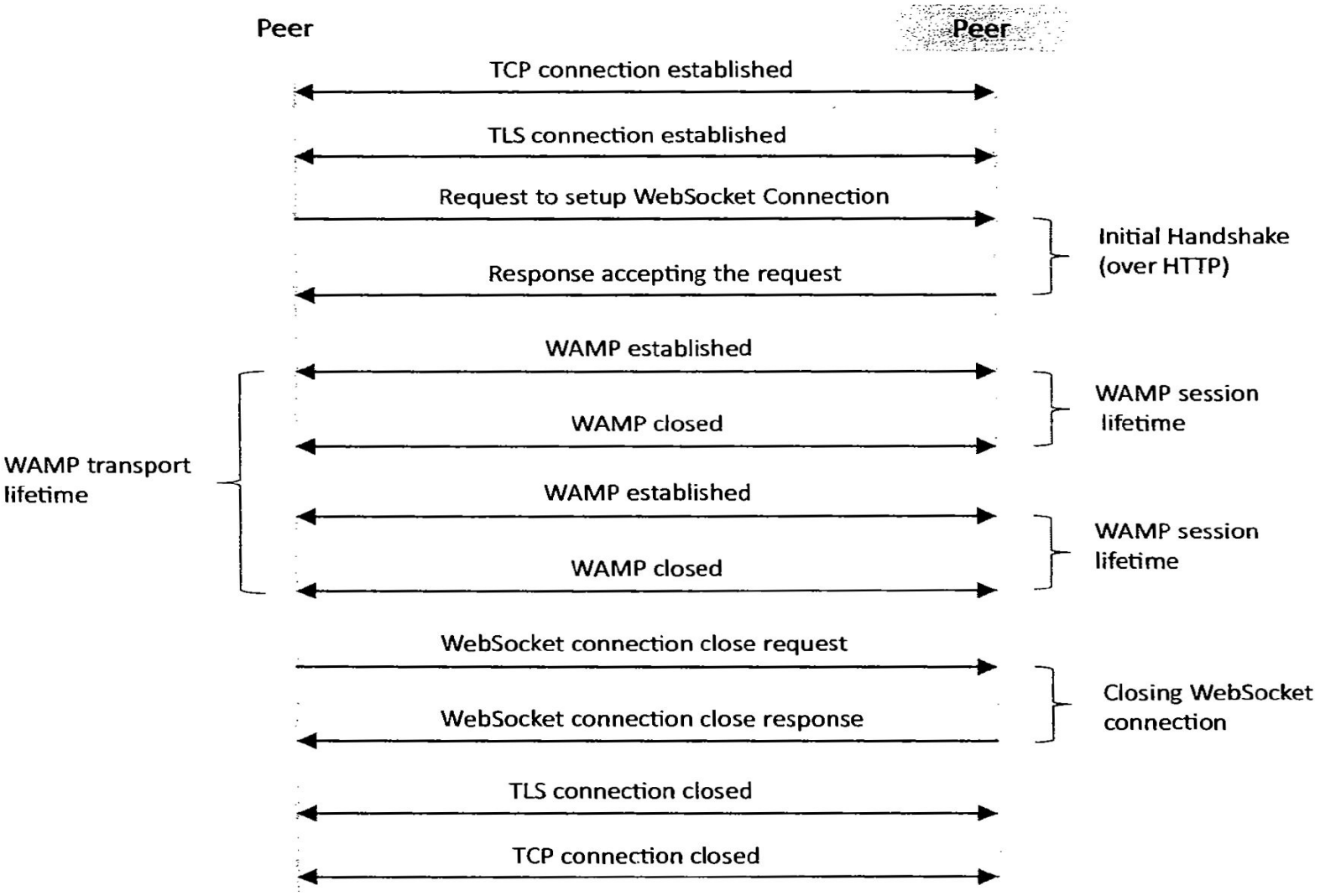
WAMP Session

Client

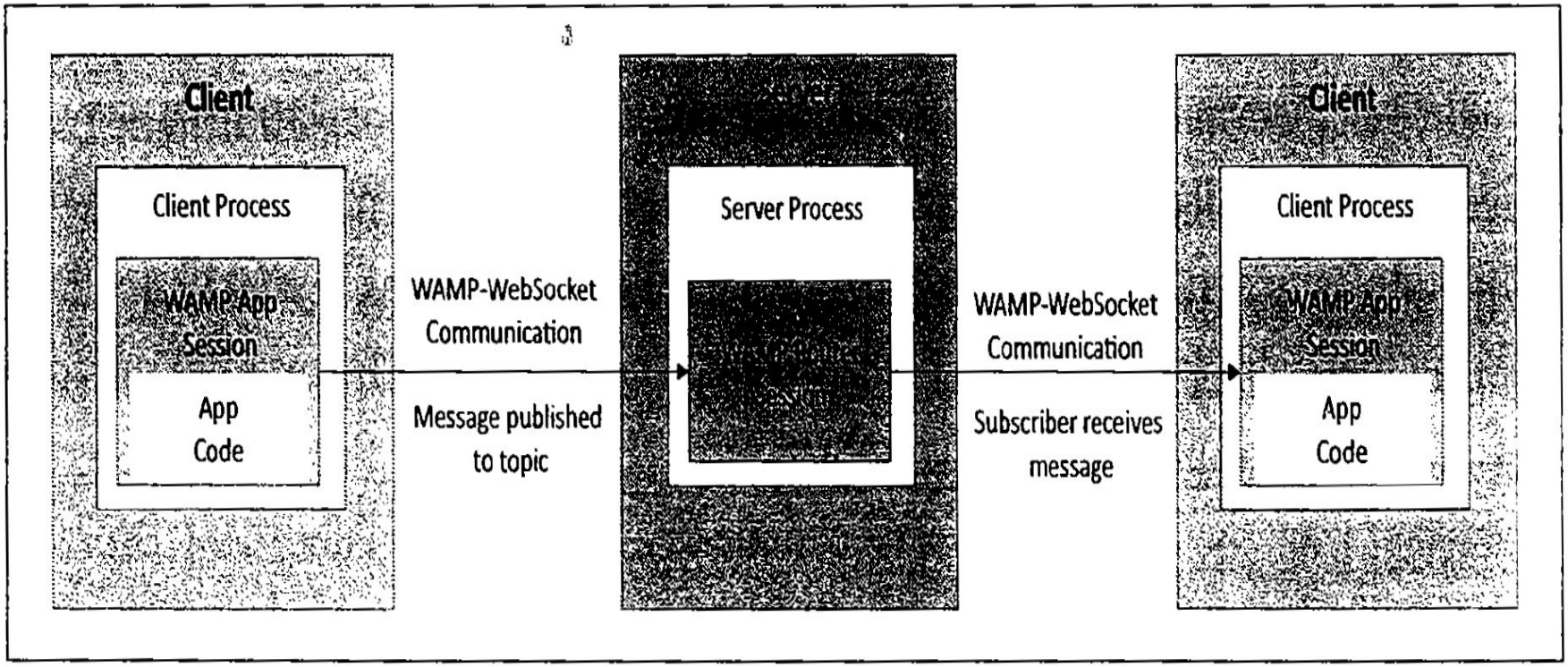
Router



WAMP Protocol



Publish-subscribe- WAMP



Conti….

- **Box 8.2: Example of a WAMP Publisher implemented using AutoBahn framework - publisherApp.py**

```
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep
from autobahn.twisted.wamp import ApplicationSession
import time,datetime

def getData():
   #Generate message
   timestamp = datetime.datetime.fromtimestamp(
```

Cont…


```
    time.time()).strftime('%Y-%m-%d%H:%M:%S')
    data = "Message at time-stamp:  "+str(timestamp)
    return data


#An application component that publishes an event every second.
class Component(ApplicationSession):
    @inlineCallbacks
    def onJoin(self, details):
        while True:
            data = getData()
            self.publish('test-topic', data)
            yield sleep(1)
```

Conti…

- **Box 8.3: Example of a WAMP Subscriber implemented using AutoBahn framework - subscriberApp.py**

```
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.wamp import ApplicationSession

#An application component that subscribes and receives events
class Component(ApplicationSession):
   @inlineCallbacks
   def onJoin(self, details):
      self.received = 0

      def on_event(data):
         print "Received message:  " + data
         yield self.subscribe(on_event, 'test-topic')

   def onDisconnect(self):
      reactor.stop()
```

Xively Cloud for IoT

- PaaS for creating solutions for IoT
- IoT developers focus on front-end infrastructure and devices for IoT
- Backend managed by Xively

Xively platform comprises of a message bus for real-time message management and routing, data services for time series archivivng ,directory services

Add Device

The Xively Developer Workbench will help you to get your devices, applications, and services talking to each other through Xively. The first step is to provision a development device. Begin by providing some basic information.

Device Name

Weather Station

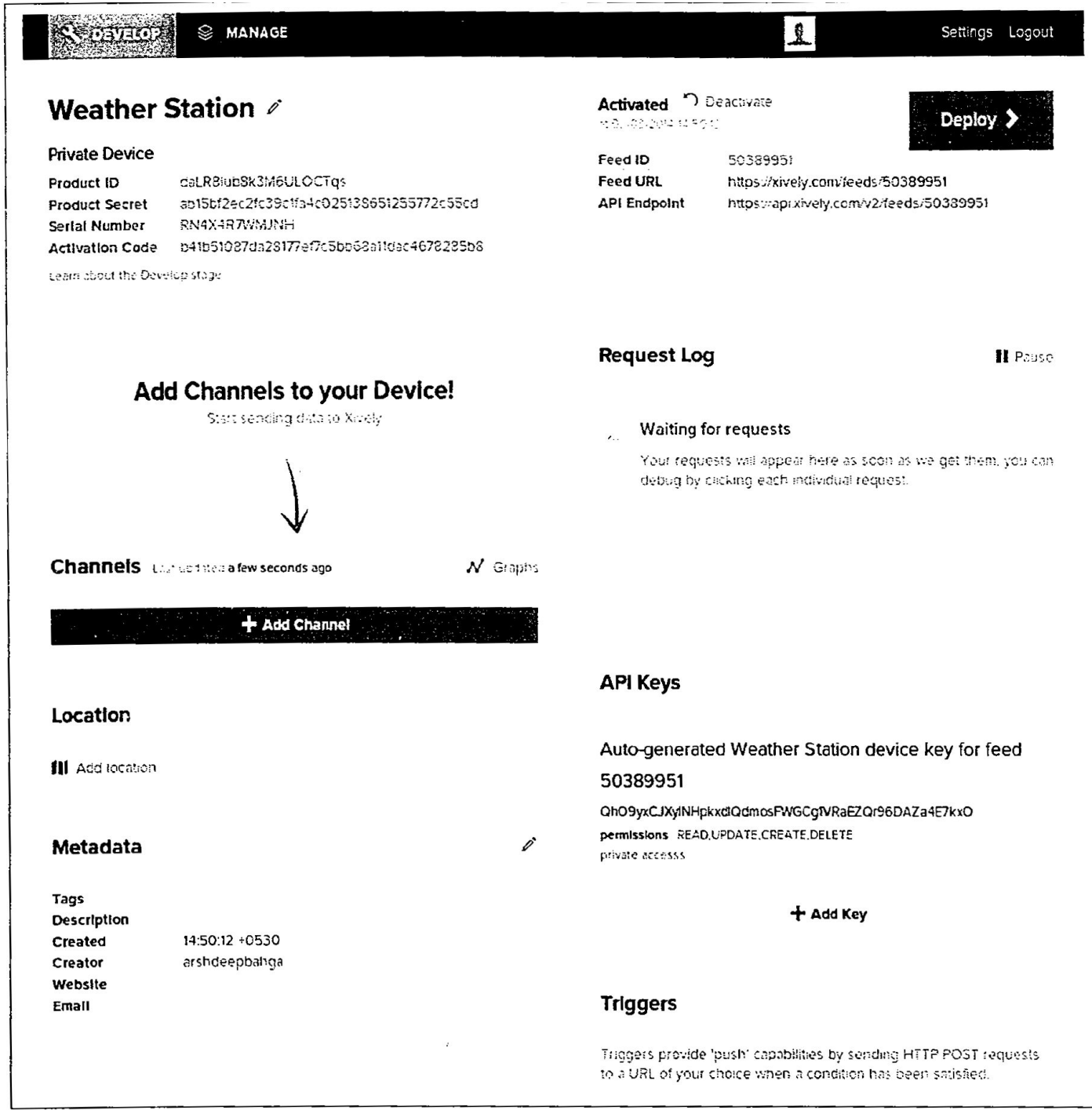
Device Description (optional)

Monitors temperature

Privacy (You own your data, we help you share it, more info)

- Private Device
  You use API keys to choose it and how you share a device's data.
- Public Device
  You agree to share a device's data under the CC0 1.0 Universal license. The Device's data is indexed by major search engines, and its Feed page is publicly viewable.

Add Device Cancel


DEVELOP
MANAGE
Settings
Logout
Weather Station
Private Device
Product ID daLR8iubSk3M6ULOCTqs
Product Secret ab15bf2ec2fc39c1fa4c025138651255772c55cd
Serial Number RN4X4R7WMJNH
Activation Code b41b51087da28177ef7c5bb68a11dac4678285b8
Learn about the Develop stage
Activated
Deactivate
Deploy
Feed ID 50389951
Feed URL https://xively.com/feeds/50389951
API Endpoint https://api.xively.com/v2/feeds/50389951
Add Channels to your Device!
Start sending data to Xively
Request Log
Pause
Waiting for requests
Your requests will appear here as soon as we get them, you can debug by clicking each individual request.
Channels
a few seconds ago
Graphs
Add Channel
Location
Add location
Metadata
Tags
Description
Created 14:50:12 +0530
Creator arshdeepbahga
Website
Email
API Keys
Auto-generated Weather Station device key for feed 50389951
QhO9yxCJXylNHpkxdQdmosFWGCgfVRaEZQr96DAZa4E7kxO
permissions READ,UPDATE,CREATE,DELETE
private accesss
Add Key
Triggers
Triggers provide 'push' capabilities by sending HTTP POST requests to a URL of your choice when a condition has been satisfied.

Box 8.4: Python program sending data to Xively Cloud

```
import time
import datetime
import requests
import xively
from random import randint
global temp_datastream
#Initialize Xively Feed
FEED_ID = "<enter feed-id>"
API_KEY = "<enter api-key>"
api = xively.XivelyAPIClient(API_KEY)

#Function to read Temperature Sensor
def readTempSensor():
    #Return random value
    return randint(20,30)

#Controller main function
def runController():
    global temp_datastream
```

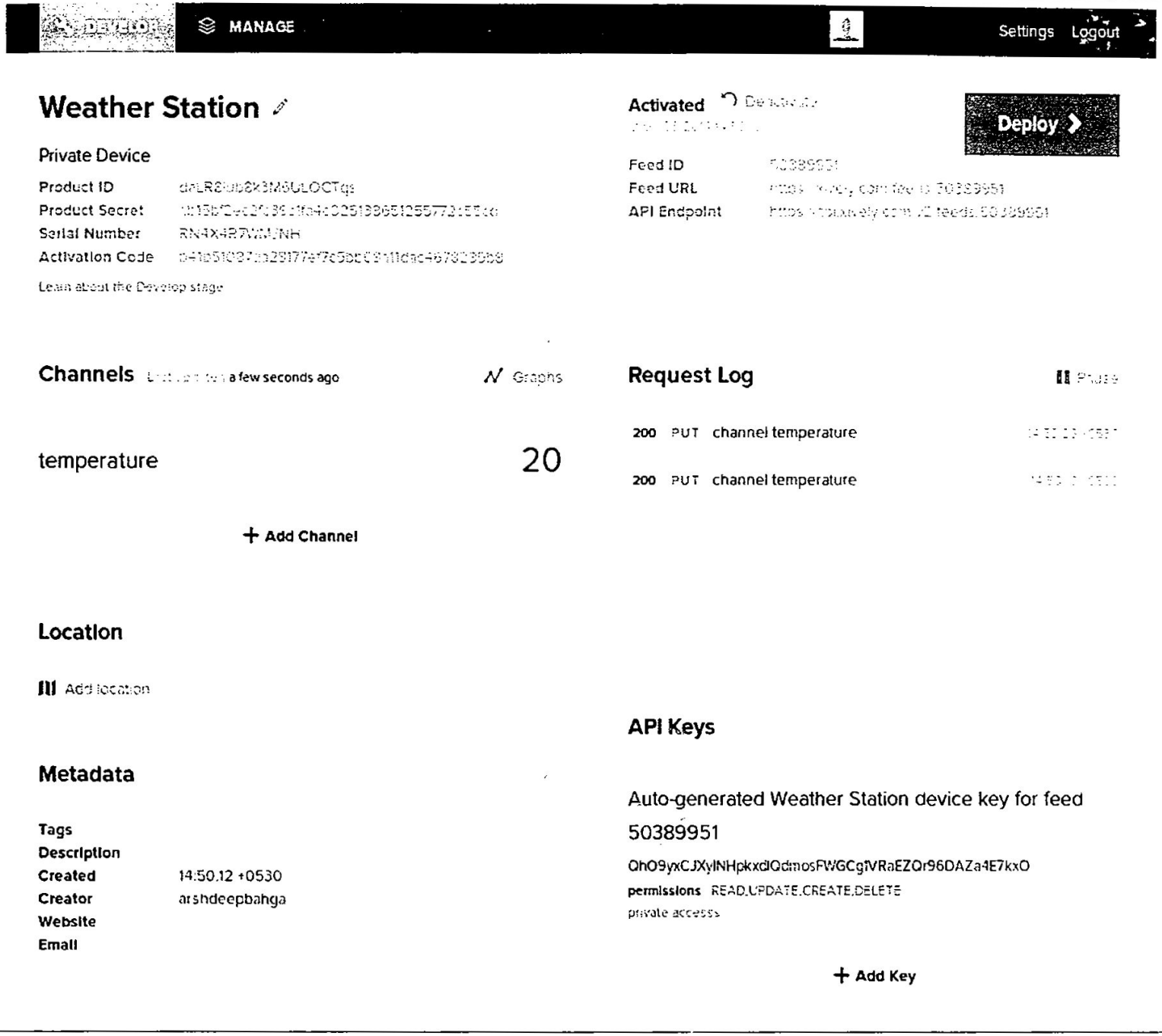
```
    temperature=readTempSensor()
    temp_datastream.current_value = temperature
    temp_datastream.at = datetime.datetime.utcnow()

    print "Updating Xively feed with Temperature:  %s" % temperature
    try:
        temp_datastream.update()
    except requests.HTTPError as e:
        print "HTTPError(0):  1".format(e.errno, e.strerror)

#Function to get existing or
#create new Xively data stream for temperature
def get_tempdatastream(feed):
    try:
        datastream = feed.datastreams.get("temperature")
        return datastream
    except:
        datastream = feed.datastreams.create("temperature",
        tags="temperature")
        return datastream

#Controller setup function
def setupController():
    global temp_datastream
    feed = api.feeds.get(FEED_ID)
    feed.location.lat="30.733315"
    feed.location.lon="76.779418"
    feed.tags="Weather"
    feed.update()

    temp_datastream = get_tempdatastream(feed)
    temp_datastream.max_value = None
    temp_datastream.min_value = None
setupController()
while True:
    runController()
    time.sleep(10)
```


DEVELOP
MANAGE
Settings
Logout
Weather Station
Private Device
Product ID
Product Secret
Serial Number
Activation Code
Learn about the Develop stage
Activated
Deploy
Feed ID
Feed URL
API Endpoint
Channels
a few seconds ago
Graphs
temperature
20
+ Add Channel
Request Log
200 PUT channel temperature
200 PUT channel temperature
Location
Add location
Metadata
Tags
Description
Created 14:50.12 +0530
Creator arshdeepbahga
Website
Email
API Keys
Auto-generated Weather Station device key for feed 50389951
permissions READ,UPDATE,CREATE,DELETE
private access
+ Add Key

Amazon Web Services for IoT

- Compute
  - EC2
  - AutoScaling
- Database
  - RDS
  - DynamoDB
- Storage and Content Delivery
  - S3
- Analytics
  - EMR
  - Kinesis
- Application Service
  - SQS

Amazon EC2 (Elastic Compute Cloud)

- EC2 – Elastic Compute Cloud
- It is an **Infrastructure-as-a-Service** (IaaS)

Scalable, pay-as-you-go compute capacity

- It is a web service that provides **computing capacity in the form of virtual machines** that are launched in Amazon's cloud computing environment

- EC2 can be **used for several purpose for IoT** systems
  - Developers can deploy IoT applications
  - Can setup IoT platforms with REST web services

Amazon EC2 (Elastic Compute Cloud)

Region:

Amazon EC2 is hosted in multiple locations world-wide. These locations are composed of Regions and Availability Zones. Each Region is a separate geographic area.

Access keys consist of two parts: an access key ID and secret access key. Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests.

AMI-Amazon Machine Image (AMI)

Amazon EC2 (Elastic Compute Cloud)

- A connection to EC2 service is firstestablished by calling boto.ec2.connect_to_region (EC2 region, AWS access key, AWS secret key)

New instance is launched using conn.run_instances function (AMI-ID, instance type, EC2 key handle, security group)

- Returns reservation
- Instances associated with reservation are obtained using reservation.instances
- Status is obtained by instance.update function

Amazon EC2 (Elastic Compute Cloud)

- The following program waits till the status of the newly launched instance becomes running and then prints the instance details such as DNS, instance IP, launch time etc.

Python program for launching an EC2 instance


```
import boto.ec2
from time import sleep
```

#EC2 interface of AWS

```
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
AMI_ID = "ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

print "Connecting to EC2"

conn = boto.ec2.connect_to_region(REGION,
            aws_access_key_id=ACCESS_KEY,
            aws_secret_access_key=SECRET_KEY)

        print "Launching instance with AMI-ID %s, with keypair
        %s, instance type %s, security group
        %s"%(AMI_ID,EC2_KEY_HANDLE,INSTANCE_TYPE,SECGROUP_HANDLE)
```

```
reservation = conn.run_instances(image_id=AMI_ID,
        key_name=EC2_KEY_HANDLE,
        instance_type=INSTANCE_TYPE,
        security_groups = [ SECGROUP_HANDLE, ] )

instance = reservation.instances[0]

print "Waiting for instance to be up and running"

status = instance.update()
while status == 'pending':
   sleep(10)
   status = instance.update()

if status == 'running':
   print " \n Instance is now running.  Instance details are:"
   print "Intance Size:  " + str(instance.instance_type)
   print "Intance State:  " + str(instance.state)
   print "Intance Launch Time:  " + str(instance.launch_time)
   print "Intance Public DNS: " + str(instance.public_dns_name)
   print "Intance Private DNS: " + str(instance.private_dns_name)
   print "Intance IP: " + str(instance.ip_address)
   print "Intance Private IP: " + str(instance.private_ip_address)
```

Python program to stopping an EC2 instance

1. Call conn.get_all_instances – returns reservations
2. Ids of instances associated with each reservation are obtained
3. conn.stop_instances to stop eachinstance

Python program to stopping an EC2 instance


```
import boto.ec2
from time import sleep


ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"

print "Connecting to EC2"

conn = boto.ec2.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Getting all running instances"
reservations = conn.get_all_instances()
print reservations
```

```
instance_rs = reservations[0].instances
instance = instance_rs[0]
instanceid=instance_rs[0].id
print "Stopping instance with ID: " + str(instanceid)

conn.stop_instances(instance_ids=[instanceid])

status = instance.update()
while not status == 'stopped':
   sleep(10)
   status = instance.update()

print "Stopped instance with ID: " + str(instanceid)
```

Amazon's Autoscaling

- Autoscaling allows **automatically scaling Amazon EC2 capacity up or down** according to user defined conditions

- Autoscaling users can **increase the number of EC2 instances running their applications** seamlessly during spikes in the application workloads **to meet the application performance requirements**

- **Scale down capacity** when the workload is low **to save costs**

Amazon S3

- S3 – Simple Storage Service
- It is an online cloud-based data storage infrastructure for storing and retrieving very large amount of data
- S3 provides highly reliable, scalable, fast, fully redundant and affordable storage infrastructure
- S3 can serve as a raw data storage

Python program for uploading a file to an S3 bucket


```
import boto.s3

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

conn = boto.connect_s3(aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

def percent_cb(complete, total):
   print ('.')

def upload_to_s3_bucket_path(bucketname, path, filename):
   mybucket = conn.get_bucket(bucketname)
   fullkeyname=os.path.join(path,filename)
   key = mybucket.new_key(fullkeyname)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

def upload_to_s3_bucket_root(bucketname, filename):
   mybucket = conn.get_bucket(bucketname)
   key = mybucket.new_key(filename)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

upload_to_s3_bucket_path('mybucket2013', 'data', 'file.txt')
```

Amazon RDS

- It is web service that allows you **to create instances of MySQL, Oracle or Microsoft SQL Server** in the cloud

- With RDS, developers **can easily setup, operate and scale a relational database** in the cloud

- RDS can serve as a **scalable data store for IoT** systems. With RDS, IoT system developers can store any amount of data in scalable relational databases

Amazon SQS

- SQS – Simple Queue Service
- Amazon SQS offers a highly scalable and reliable hosted queue for storing message as they travel between distinct components of applications
- SQS guarantees only that message arrive, not that they arrive in the same order in which they were put in the queue
- It is simply a queue system that store and releases messages in a scalable manner
- It can be used in distributed IoT applications in which various application components need to exchange messages