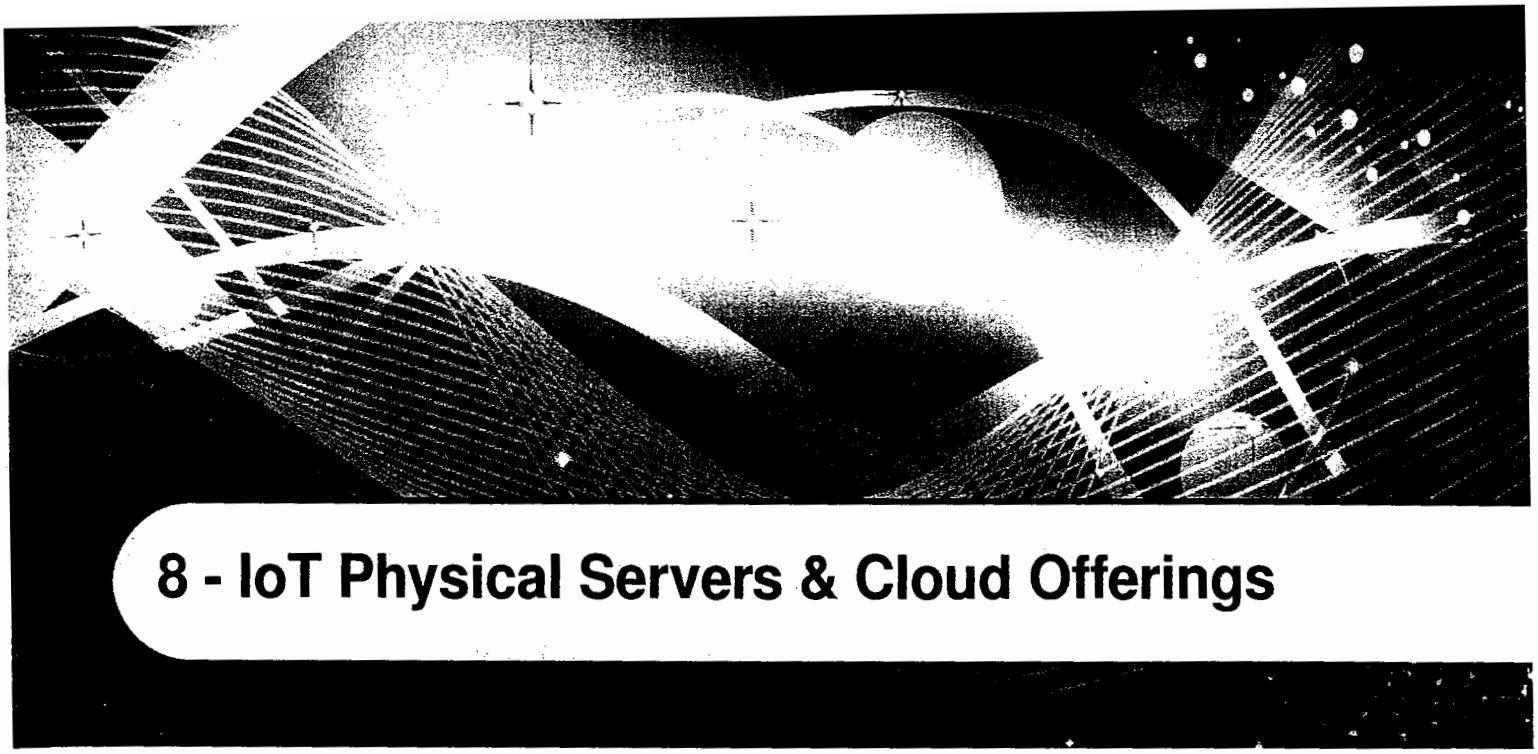

8 - IoT Physical Servers & Cloud Offerings

This Chapter Covers

- Cloud Storage Models & Communication APIs
- Web Application Messaging Protocol (WAMP)
- Xively cloud for IoT
- Python web application framework - Django
- Developing applications with Django
- Developing REST web services
- Amazon Web Services for IoT
- SkyNet IoT Messaging Platform

8.1 Introduction to Cloud Storage Models & Communication APIs

Cloud computing is a transformative computing paradigm that involves delivering applications and services over the Internet. NIST defines cloud computing as [77] - Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. The interested reader may want to refer to the companion book on Cloud Computing by your authors.

In this chapter you will learn how to use cloud computing for Internet of Things (IoT). You will learn about the Web Application Messaging Protocol (WAMP), Xively's Platform-as-a-Service (PaaS) which provides tools and services for developing IoT solutions. You will also learn about the Amazon Web Services (AWS) and their applications for IoT.

8.2 WAMP - AutoBahn for IoT

Web Application Messaging Protocol (WAMP) is a sub-protocol of Websocket which provides publish-subscribe and remote procedure call (RPC) messaging patterns. WAMP enables distributed application architectures where the application components are distributed on multiple nodes and communicate with messaging patterns provided by WAMP.

Let us look at the key concepts of WAMP:

- **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.
- **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).

Figure 8.1 shows a WAMP Session between Client and Router, established over a Transport. Figure 8.2 shows the WAMP protocol interactions between peers. In this figure the WAMP transport used is WebSocket. Recall the WebSocket protocol diagram explained in Chapter-1. WAMP sessions are established over WebSocket transport within the lifetime of WebSocket transport.

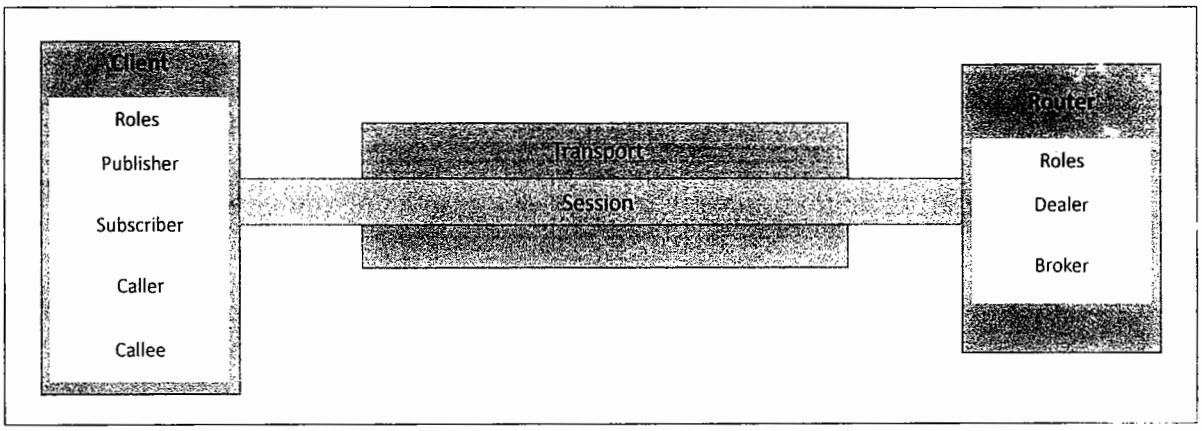

Figure 8.1: WAMP Session between Client and Router

For the examples in this hands-on book we use the AutoBahn framework which provides open-source implementations of the WebSocket and WAMP protocols [100].

Figure 8.3 shows the communication between various components of a typical WAMP-AutoBahn deployment. The Client (in Publisher role) runs a WAMP application component that publishes messages to the Router. The Router (in Broker role) runs on the Server and routes the messages to the Subscribers. The Router (in Broker role) decouples the Publisher from the Subscribers. The communication between Publisher - Broker and Broker - Subscribers happens over a WAMP-WebSocket session.

Let us look at an example of a WAMP publisher and subscriber implemented using AutoBahn. Box 8.1 shows the commands for installing AutoBahn-Python.

■ **Box 8.1: Commands for installing AutoBahn**

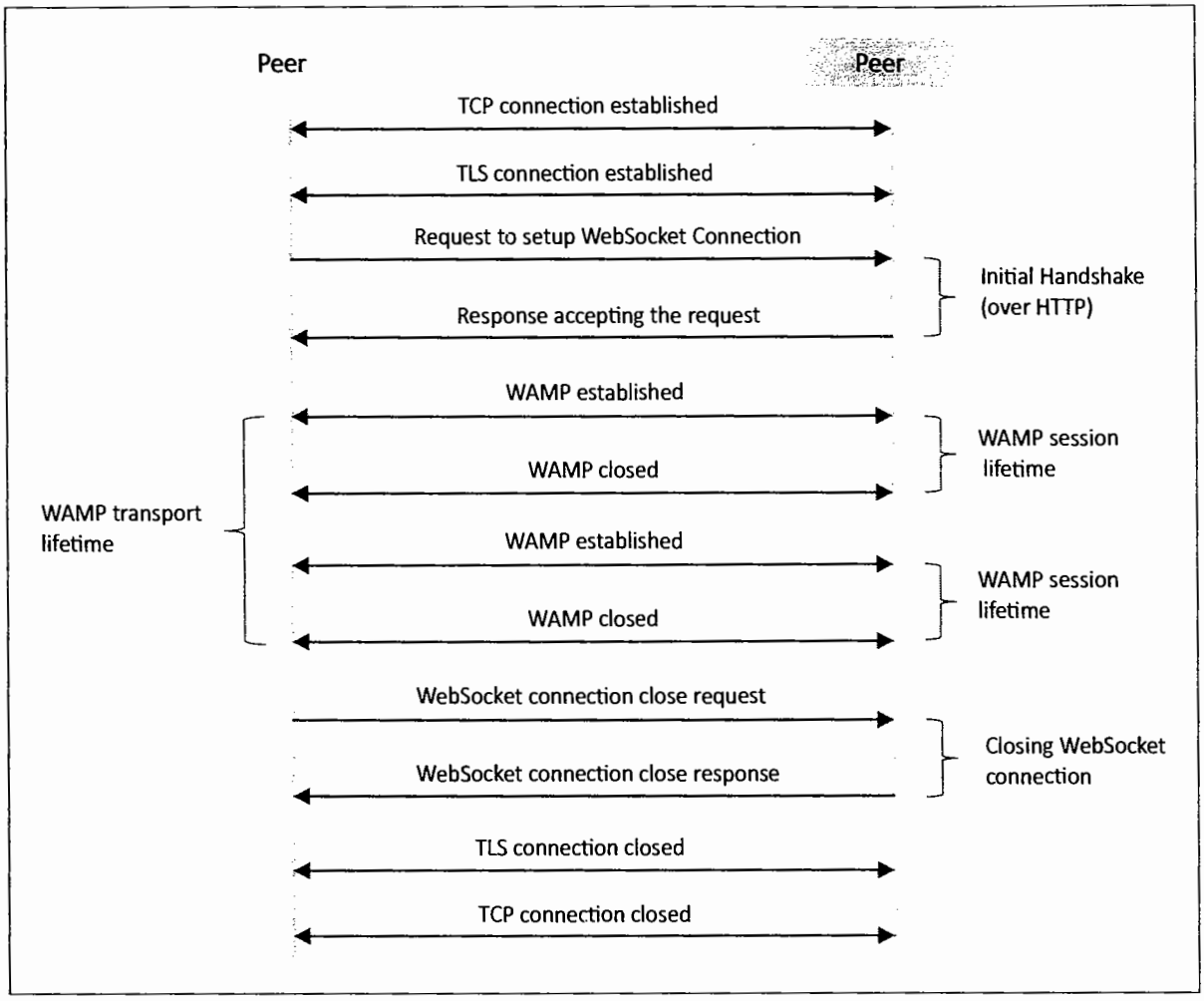

Figure 8.2: WAMP protocol

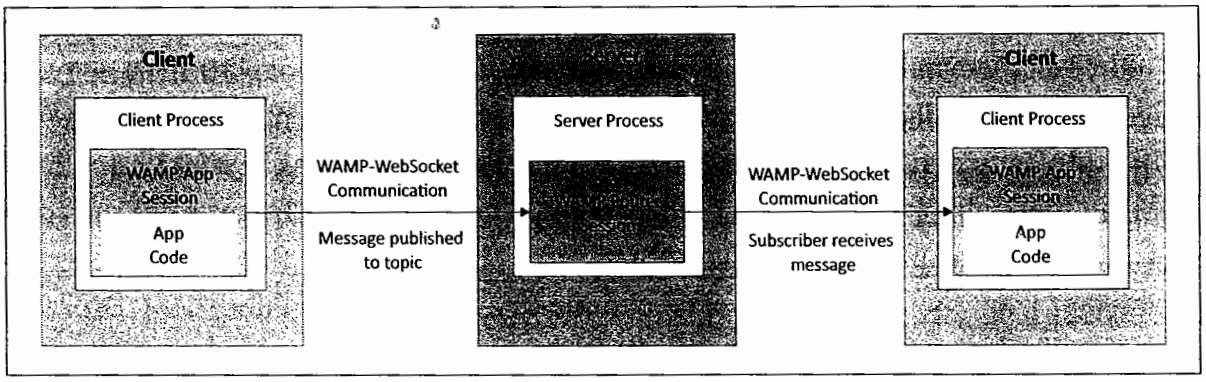

Figure 8.3: Publish-subscribe messaging using WAMP-AutoBahn

```
#Setup Autobahn
sudo apt-get install python-twisted python-dev
```

```
sudo apt-get install python-pip
sudo pip install -upgrade twisted
sudo pip install -upgrade autobahn
```

After installing AutoBahn, clone AutobahnPython from GitHub as follows:

- git clone https://github.com/tavendo/AutobahnPython.git

Create a WAMP publisher component as shown in Box 8.2. The publisher component publishes a message containing the current time-stamp to a topic named 'test-topic'. Next, create a WAMP subscriber component as shown in Box 8.3. The subscriber component that subscribes to the 'test-topic'. Run the application router on a WebSocket transport server as follows:

- python AutobahnPython/examples/twisted/wamp/basic/server.py

Run the publisher component over a WebSocket transport client as follows:

- python AutobahnPython/examples/twisted/wamp/basic/client.py –component "publisherApp.Component"

Run the subscriber component over a WebSocket transport client as follows:

- python AutobahnPython/examples/twisted/wamp/basic/client.py –component "subscriberApp.Component"

■ 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(
```

```
    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)
```

■ 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()
```

While you can setup the server and client processes on a local machine for trying out the publish-subscribe example, in production environment, these components run on separate machines. The server process (the brains or the "Thing Tank"!) is setup on a cloud-based instance while the client processes can run either on local hosts/devices or in the cloud.

8.3 Xively Cloud for IoT

Xively is a commercial Platform-as-a-Service that can be used for creating solutions for Internet of Things. With Xively cloud, IoT developers can focus on the front-end

infrastructure and devices for IoT (that generate the data), while the backend data collection infrastructure is managed by Xively.

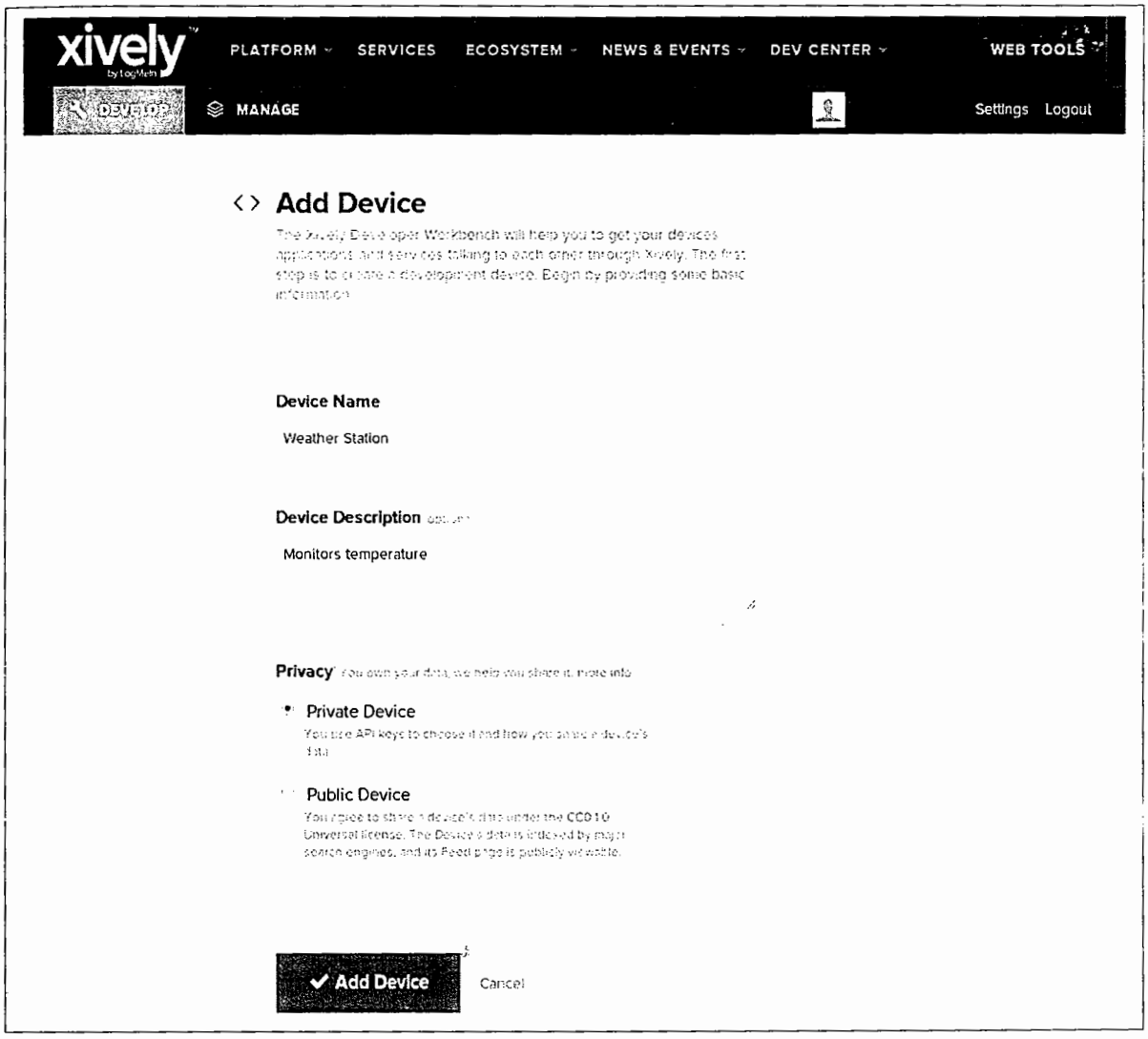

Figure 8.4: Screenshot of Xively dashboard - creating a new device

Xively platform comprises of a message bus for real-time message management and routing, data services for time series archiving, directory services that provides a search-able directory of objects and business services for device provisioning and management. Xively provides an extensive support for various languages and platforms. The Xively libraries leverage standards-based API over HTTP, Sockets and MQTT for connecting IoT devices to the Xively cloud. In this chapter we will describe how to use the Xively Python library.

To start using Xively, you have to register for a developer account. You can then create development devices on Xively. Figures 8.4 shows screenshot of how to create a new device from the Xively dashboard. When you create a device, Xively automatically creates a

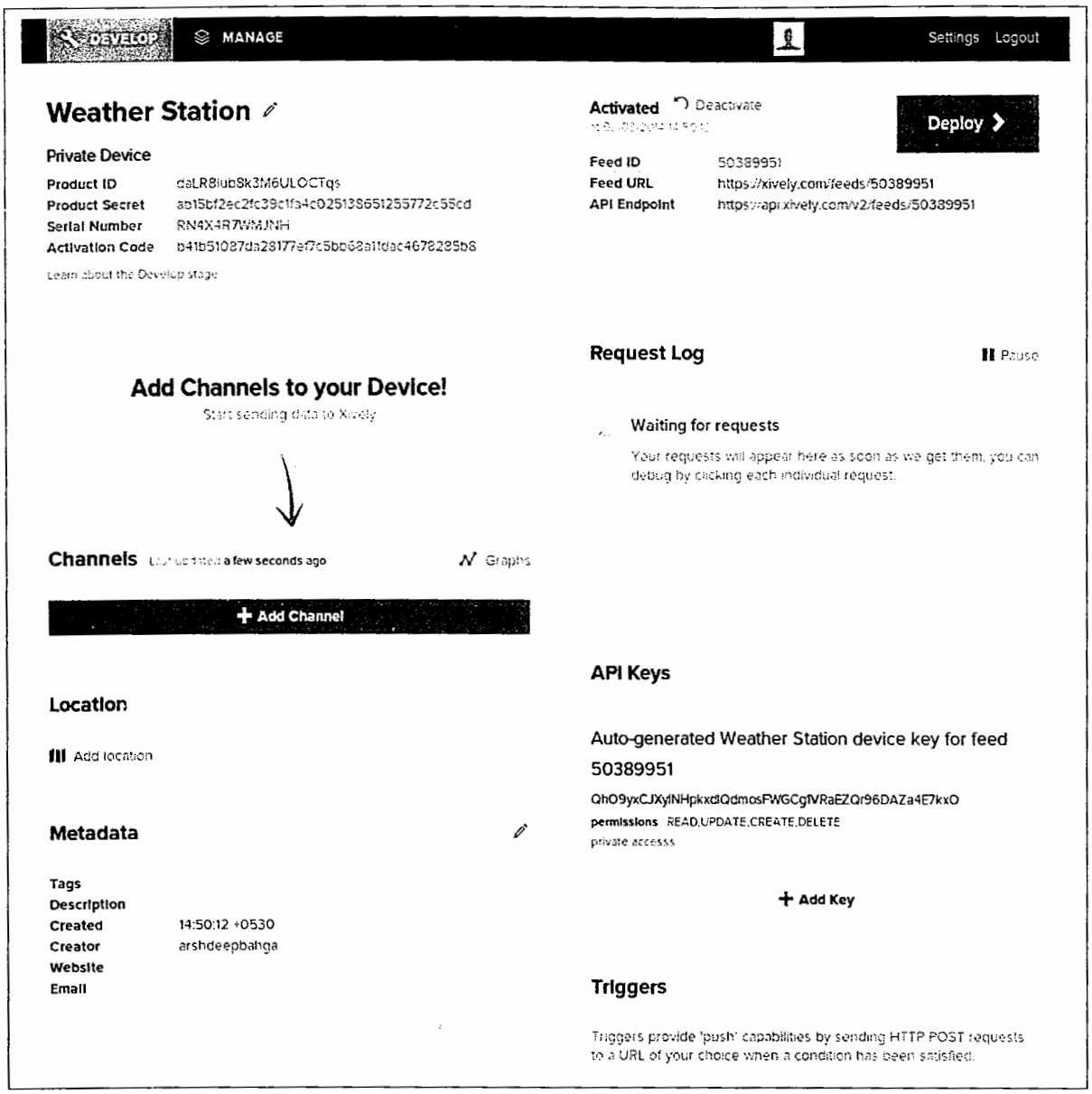

Figure 8.5: Screenshot of Xively dashboard - device details

Feed-ID and an API Key to connect to the device as shown in Figures 8.5. Each device has a unique Feed-ID. Feed-ID is a collection of channels or datastreams defined for a device and the associated meta-data. API keys are used to provide different levels of permissions. The default API key has read, update, create and delete permissions.

Xively devices have one or more channels. Each channel enables bi-directional communication between the IoT devices and the Xively cloud. IoT devices can send data to a channel using the Xively APIs. For each channel, you can create one or more triggers. A trigger specification includes a channel to which the trigger corresponds, trigger condition

(e.g. channel value less than or greater than a certain value) and an HTTP POST URL to which the request is sent when the trigger fires. Triggers are used for integration with third-party applications.

Let us look at an example of using Xively cloud for an IoT system that monitors temperature and sends the measurements to a Xively channel. The temperature monitoring device can be built with the Raspberry Pi board and a temperature sensor connected to the board. The Raspberry Pi runs a controller program that reads the sensor values every few seconds and sends the measurements to a Xively channel. Box 8.4 shows the Python program for the sending temperature data to Xively Cloud. This example uses the Xively Python library. To keep the program simple and without going into the details of the temperature sensor we use synthetic data (generated randomly in *readTempSensor*() function). The complete implementation of the *readTempSensor*() function is described in the next chapter. In this controller program, a feed object is created by providing the API key and Feed-ID. Then a channel named *temperature* is created (if not existing) or retrieved. The temperature data is sent to this channel in the *runController*() function every 10 seconds. Figures 8.6 shows the temperature channel in the Xively dashboard. In this example we created a single Xively device with one channel. In real-world scenario each Xively device can have multiple channels and you can have multiple devices in a production batch.

■ 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)
```

8.4 Python Web Application Framework - Django

In the previous section, you learned about the Xively PaaS for collecting and processing data from IoT systems in the cloud. You learned how to use the Xively Python library. To build IoT applications that are a backed by Xively cloud or any other data collection systems, you

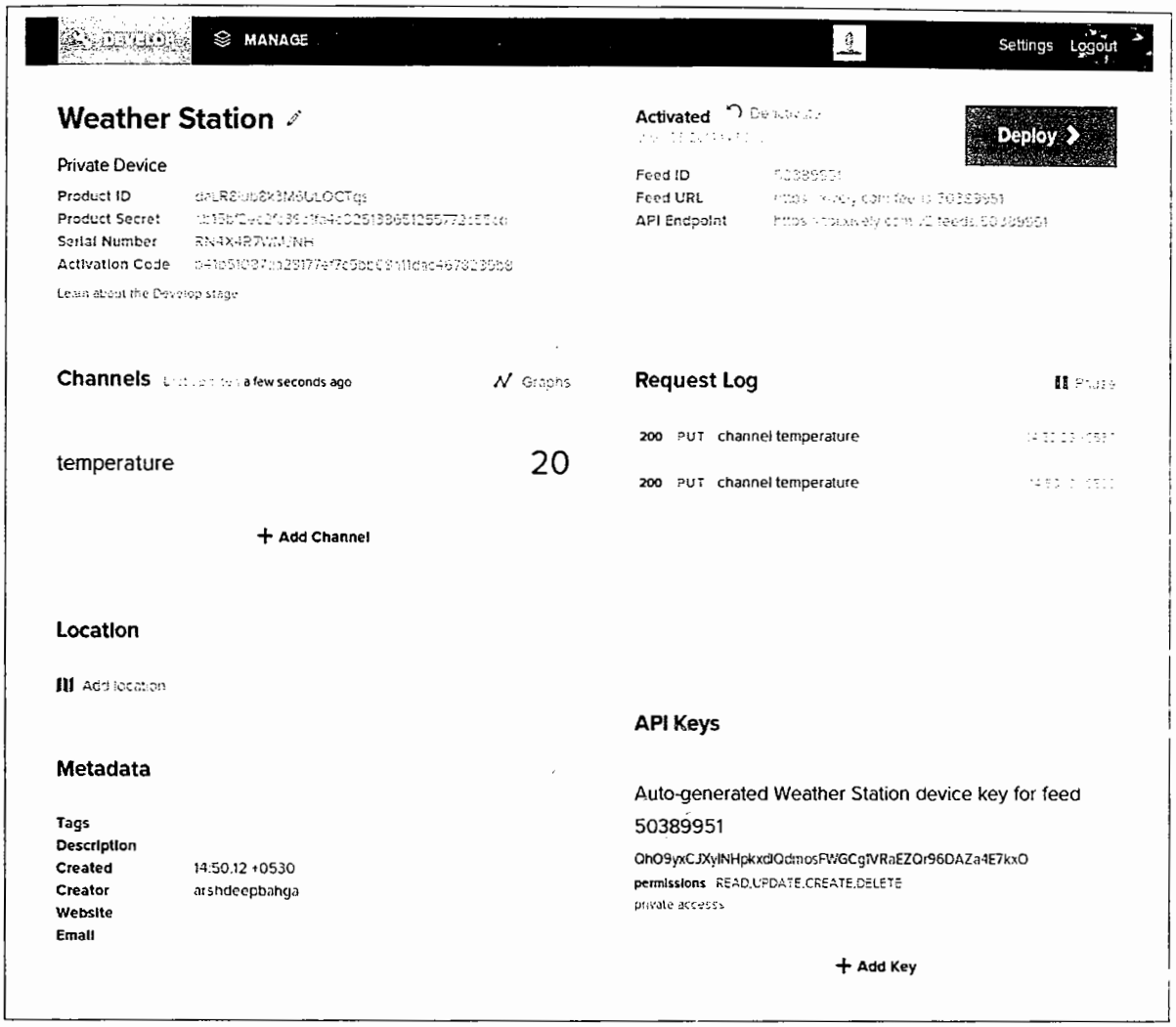

Figure 8.6: Screenshot of Xively dashboard - data sent to channel

would require some type of web application framework. In this section you will learn about a Python-based web application framework called Django.

Django is an open source web application framework for developing web applications in Python [116]. A "web application framework" in general is a collection of solutions, packages and best practices that allows development of web applications and dynamic websites. Django is based on the well-known Model-Template-View architecture and provides a separation of the data model from the business rules and the user interface. Django provides a unified API to a database backend. Therefore, web applications built with Django can work with different databases without requiring any code changes. With this flexibility in web application design combined with the powerful capabilities of the Python language and the Python ecosystem, Django is best suited for IoT applications. Django, concisely stated, consists of an object-relational mapper, a web templating system and a

regular-expression-based URL dispatcher.

8.4.1 Django Architecture

Django is a Model-Template-View (MTV) framework wherein the roles of model, template and view, respectively, are:

Model

The model acts as a definition of some stored data and handles the interactions with the database. In a web application, the data can be stored in a relational database, non-relational database, an XML file, etc. A Django model is a Python class that outlines the variables and methods for a particular type of data.

Template

In a typical Django web application, the template is simply an HTML page with a few extra placeholders. Django's template language can be used to create various forms of text files (XML, email, CSS, Javascript, CSV, etc.)

View

The view ties the model to the template. The view is where you write the code that actually generates the web pages. View determines what data is to be displayed, retrieves the data from the database and passes the data to the template.

8.4.2 Starting Development with Django

Appendix C provides the instructions for setting up Django. In this section you will learn how to start developing web applications with Django.

Creating a Django Project and App

Box 8.5 provides the commands for creating a Django project and an application within a project.

When you create a new django project a number of files are created as described below:

- __init__.py: This file tells Python that this folder is a Python package
- manage.py: This file contains an array of functions for managing the site.
- settings.py: This file contains the website's settings
- urls.py: This file contains the URL patterns that map URLs to pages.

A Django project can have multiple applications ("apps"). Apps are where you write the code that makes your website function. Each project can have multiple apps and each app can be part of multiple projects.

When a new application is created a new directory for the application is also created which has a number of files including:

- model.py: This file contains the description of the models for the application.
- views.py: This file contains the application views.

Box 8.5: Creating a new Django project and an app in the project

```
#Create a new project
django-admin.py startproject blogproject

#Create an application within the project
python mangage.py startapp myapp

#Starting development server
python manage.py runserver

#Django uses port 8000 by default
#The project can be viewed at the URL:
#http://localhost:8000
```

Django comes with a built-in, lightweight Web server that can be used for development purposes. When the Django development server is started the default project can be viewed at the URL: http://localhost:8000. Figure 8.7 shows a screenshot of the default project.

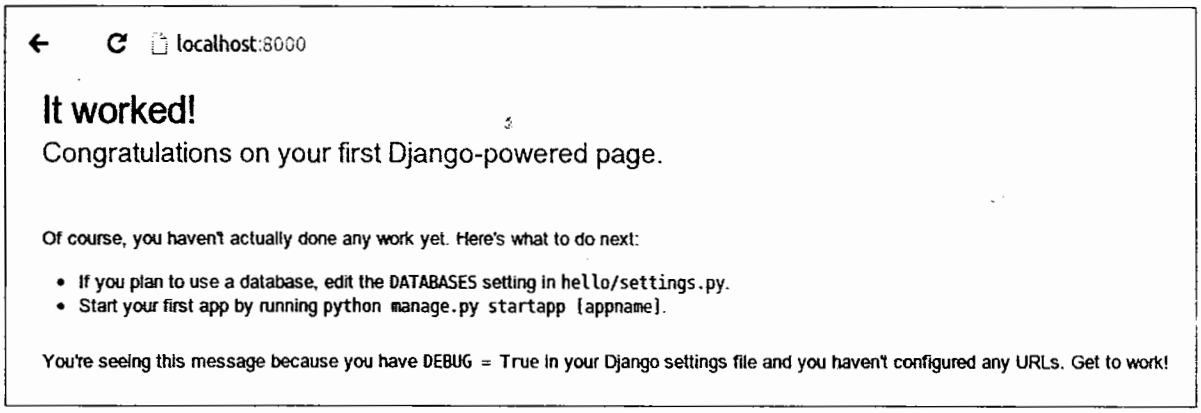

Figure 8.7: Django default project

Configuring a Database

Till now you have learned how to create a new Django project and an app within the project. Most web applications have a database backend. Developers have a wide choice of

databases that can be used for web applications including both relational and non-relational databases. Django provides a unified API for database backends thus giving the freedom to choose the database. Django supports various relational database engines including MySQL, PostgreSQL, Oracle and SQLite3. Support for non-relational databases such as MongoDB can be added by installing additional engines (e.g. Django-MongoDB engine for MongoDB).

Let us look at examples of setting up a relational and a non-relational database with a Django project. The first step in setting up a database is to install and configure a database server. After installing the database, the next step is to specify the database settings in the setting.py file in the Django project.

Box 8.6 shows the commands to setup MySQL. Box 8.7 shows the database setting to use MySQL with a Django project.

■ Box 8.6: Setting up MySQL database

```
#Install MySQL
sudo apt-get install mysql-server mysql-client
sudo mysqladmin -u root -h localhost password 'mypassword'
```

■ Box 8.7: Configuring MySQL with Django - settings.py

```
DATABASES = {
   'default':  {
      'ENGINE': 'django.db.backends.mysql',
      'NAME': '<database-name>'
      'USER': 'root'
      'PASSWORD': 'mypassword'
      'HOST': '<hostname>', # set to empty for localhost
      'PORT': '<port>', #set to empty for default port
}
}
```

Box 8.8 shows the commands to setup MongoDB and the associated Django-MongoDB engine. Box 8.9 shows the database setting to use MongoDB within a Django project.

■ Box 8.8: Setting up MongoDB and Django-MongoDB engine

```
#Install MongoDB
sudo apt-key adv -keyserver keserver.ubuntu.com -recv 7F0CEB10
```

```
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen' | sudo tee /etc/apt/sources.list.d/10gen.list

sudo apt-get update
sudo apt-get install mongodb-10gen

#Setup Django MongoDB Engine
sudo pip install
https://bitbucket.org/wkornewald/django-nonrel/get/tip.tar.gz
sudo pip install
https://bitbucket.org/wkornewald/djangotoolbox/get/tip.tar.gz
sudo pip install
https://github.com/django-nonrel/mongodb-engine/tarball/master
```

■ Box 8.9: Configuring MongoDB with Django - settings.py

```
DATABASES = {
   'default':  {
      'ENGINE': 'django_mongodb_engine',
      'NAME': '<database-name>'
      'HOST': '<mongodb-hostname>', # set to empty for localhost
      'PORT': '<mongodb-port>', #set to empty for default port
}
}
```

Defining a Model

A Model acts as a definition of the data in the database. In this section we will explain Django with the help of a weather station application that displays the temperature data collected by an IoT device. Box 8.10 shows an example of a Django model for *TemperatureData*. The *TemperatureData* table in the database is defined as a Class in the Django model.

Each class that represents a database table is a subclass of *django.db.models.Model* which contains all the functionality that allows the models to interact with the database. The *TemperatureData* class has fields timestamp, temperature, lat and lon all of which are *CharField*. To sync the models with the database simply run the following command:
>python manage.py syncdb

When the *syncdb* command is run for the first time, it creates all the tables defined in the Django model in the configured database. For more information about the Django models refer to the Django documentation [117].

■ Box 8.10: Example of a Django model

```
from django.db import models

class TemperatureData(models.Model):
   timestamp = models.CharField(max_length=10)
   temperature = models.CharField(max_length=5)
   lat = models.CharField(max_length=10)
   lon = models.CharField(max_length=10)
   def __unicode__(self):
      return self.timestamp
```

Django Admin Site

Django provides an administration system that allows you to manage the website without writing additional code. This "admin" system reads the Django model and provides an interface that can be used to add content to the site. The Django admin site is enabled by adding *django.contrib.admin* and *django.contrib.admindocs* to the INSTALLED_APPS section in the settings.py file. The admin site also requires URL pattern definitions in the urls.py file described later in the URLs sections.

To define which of your application models should be editable in the admin interface, a new file named admin.py is created in the application folder as shown in Box 8.11.

■ Box 8.11: Enabling admin for Django models

```
from django.contrib import admin
from myapp.models import TemperatureData

admin.site.register(TemperatureData)
```

Figure 8.8 shows a screenshot of the default admin interface. You can see all the tables corresponding to the Django models in this screenshot. Figure 8.9 shows how to add new items in the TemperatureData table using the admin site.

Defining a View

The View contains the logic that glues the model to the template. The view determines the data to be displayed in the template, retrieves the data from the database and passes it to the template. Conversely, the view also extracts the data posted in a form in the template and

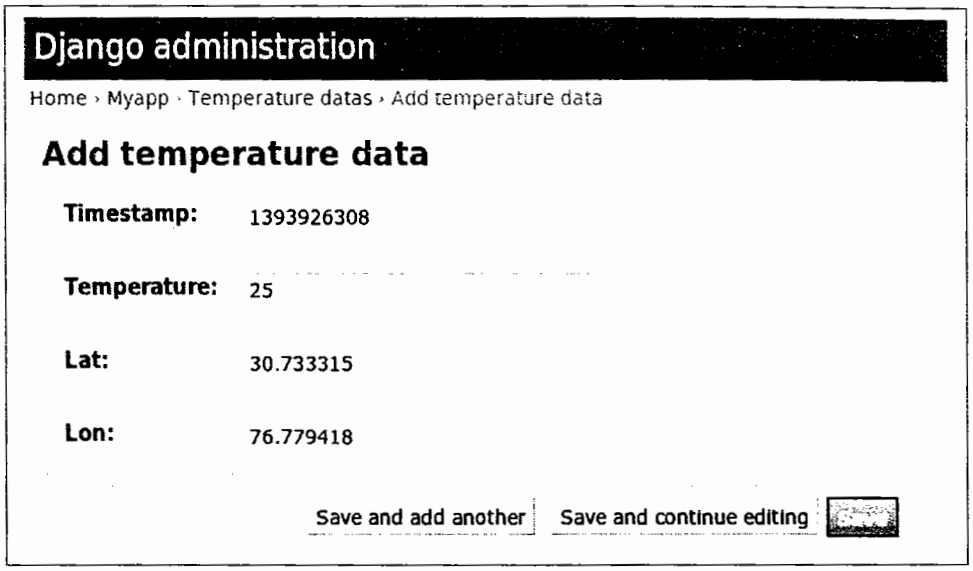

Figure 8.8: Screenshot of default Django admin interface

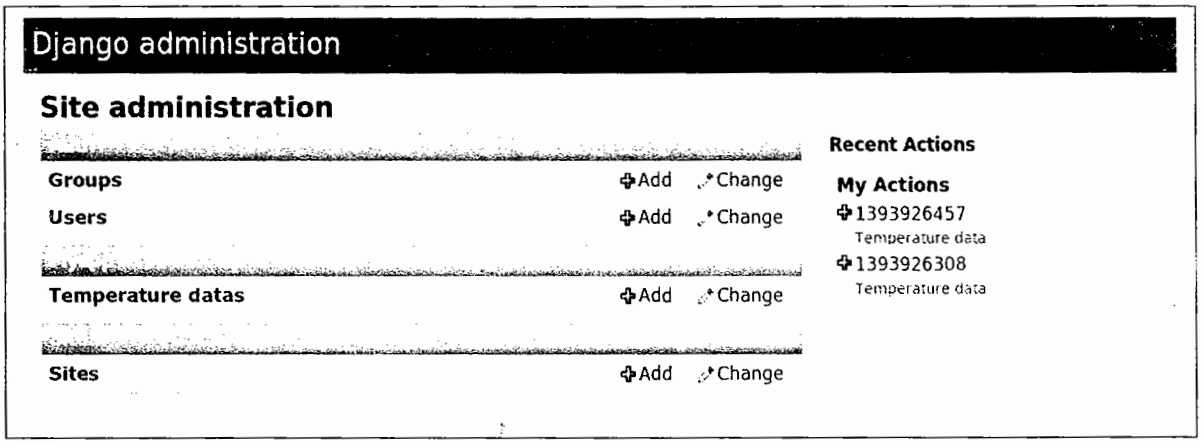

Figure 8.9: Adding data to table from Django admin interface

inserts it in the database. Typically, each page in the website has a separate view, which is basically a Python function in the views.py file. Views can also perform additional tasks such as authentication, sending emails, etc.

Box 8.12 shows an example of a Django view for the Weather Station app. This view corresponds to the webpage that displays latest entry in the TemperatureData table. In this view the Django's built in object-relational mapping API is used to retrieve the data from the TemperatureData table. The object-relational mapping API allows the developers to write generic code for interacting with the database without worrying about the underlying database engine. So the same code for database interactions works with different database backends. You can optionally choose to use a Python library specific to the database

backend used (e.g. MySQLdb for MYSQL, PyMongo for MongoDB, etc.) to write database backed specific code. For more information about the Django views refer to the Django documentation [118].

In the view shown in Box 8.12, the *TemperatureData.objects.order_by('-id')[0]* query returns the latest entry in the table. To retrieve all entries, you can use *table.objects.all()*. To retrieve specific entries, you can use *table.objects.filter(**kwargs)* to filter out queries that match the specified condition. To render the retrieved entries in the template, the *render_to_response* function is used. This function renders a given template with a given context dictionary and returns an *HttpResponse* object with that rendered text. Box 8.13 shows an alternative view that retrieves data from the Xively cloud.

■ Box 8.12: Example of a Django view

```
from django.shortcuts import render_to_response
from myapp.models import *
from django.template import RequestContext

def home(request):
   tempData = TemperatureData.objects.order_by('-id')[0]
   temperature = tempData.temperature
   lat = tempData.lat
   lon = tempData.lon

   return render_to_response('index.html','temperature':temperature,
   'lat': lat, 'lon': lon, context_instance=RequestContext(request))
```

■ Box 8.13: Alternative Django View that retrieves data from Xively

```
from django.shortcuts import render_to_response
from django.template import RequestContext
import requests
import xively

FEED_ID = "<enter-id>"
API_KEY = "<enter-key>"
api = xively.XivelyAPIClient(API_KEY)

feed = api.feeds.get(FEED_ID)
temp_datastream = feed.datastreams.get("temperature")
```

```
def home(request):
   temperature=temp_datastream.current_value
   lat=feed.location.lat
   lon=feed.location.lon

   return render_to_response('index.html','temperature':temperature,
   'lat': lat, 'lon': lon, context_instance=RequestContext(request))
```

Defining a Template

A Django template is typically an HTML file (though it can be any sort of text file such as XML, email, CSS, Javascript, CSV, etc.). Django templates allow separation of the presentation of data from the actual data by using placeholders and associated logic (using template tags). A template receives a context from the view and presents the data in context variables in the placeholders. Box 8.14 shows an example of a template for the Weather Station app. In the previous section you learned how the data is retrieved from the database in the view and passed to the template in the form of a context dictionary. In the example shown in Box 8.14, the variables containing the retrieved temperature, latitude and longitude are passed to the template. For more information about the Django templates refer to the Django documentation [119].

Box 8.14: Example of a Django template

```
<html>
<head>
<meta charset="utf-8">
<link href="/static/css/bootstrap-responsive.css" rel="stylesheet">
<script type="text/javascript"
src="http://maps.google.com/maps/api/js?sensor=false"></script>
<script type="text/javascript">
function initialize()
   var latlng = new google.maps.LatLng(lat,lon);
   var settings =
   zoom: 11,
   center: latlng,
   mapTypeControl: false,
   mapTypeControlOptions: style:
google.maps.MapTypeControlStyle.DROPDOWN_MENU,
   navigationControl: true,
   navigationControlOptions: style:
```

```
google.maps.NavigationControlStyle.SMALL,
   mapTypeId:  google.maps.MapTypeId.TERRAIN
   ;

var map = new google.maps.Map(document.getElementById("map_canvas"),
settings);
var wespiMarker = new google.maps.Marker(
   position:  latlng,
   map:  map,
   title:"CityName, "
   );
   startws ();

</script>
<title>Weather Station</title>
</head>
<body onload="initialize()">

<div class="container">
   <center><h1>Weather Station</h1></center>
   <h3>CityName</h3>
   <div class='row'>
   <div class='span3'><h4>Temperature</h4></div>
   <div class='span3'>
<h4 id='temperature'>{{temperature}}</h4></div></div>
   <div class="span6" style="height:435px">
   <div id = "map_canvas">
   </div>
   </div>
</div>

</body>
</html>
```

Figure 8.10 shows the home page for the Weather Station app. The home page is rendered from the template shown in Box 8.14.

Defining the URL Patterns

URL Patterns are a way of mapping the URLs to the views that should handle the URL requests. The URLs requested by the user are matched with the URL patterns and the view corresponding to the pattern that matches the URL is used to handle the request. Box 8.15 shows an example of the URL patterns for the Weather Station project. As seen in this example, the URL patterns are constructed using regular expressions. The simplest regular

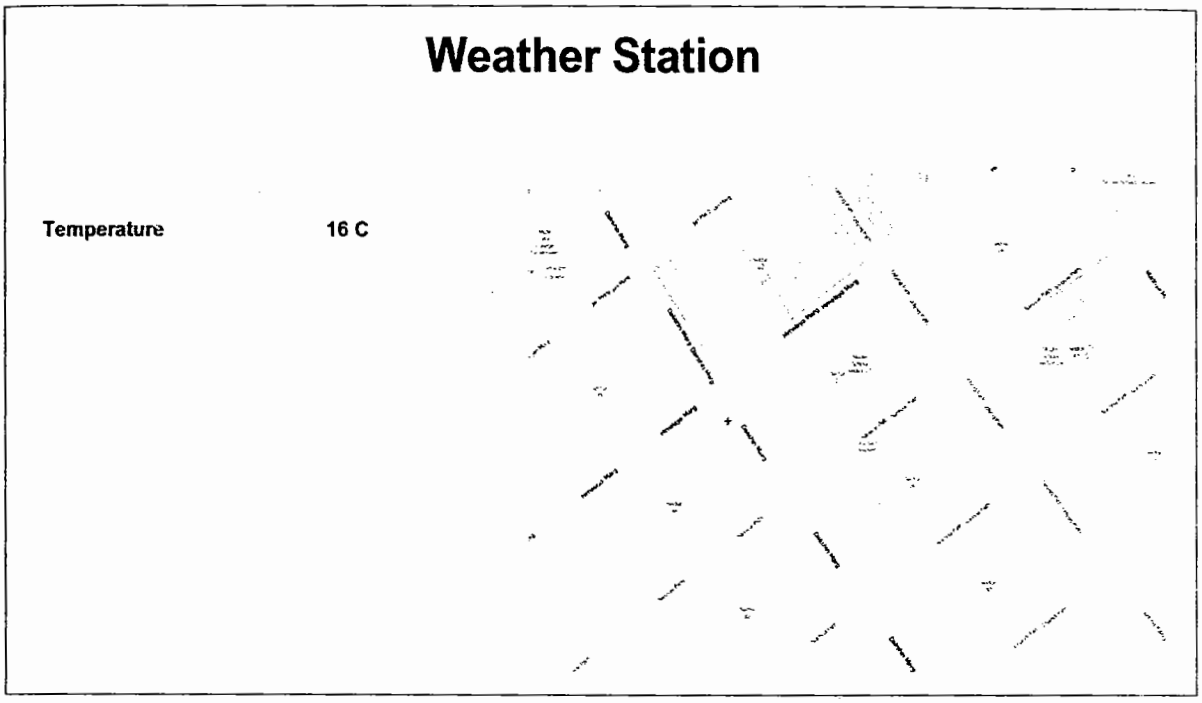

Figure 8.10: Screenshot of a temperature monitoring web application

expression (r' ^ $') corresponds to the root of the website or the home page. For more information about the Django URL patterns refer to the Django documentation [120].

■ Box 8.15: Example of a URL configuration

```
from django.conf.urls.defaults import *
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
   url(r'^$', 'myapp.views.home', name='home'),
   url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
   url(r'^admin/', include(admin.site.urls)),
)
```

With the models, views, templates and URL patterns defined for the Django project, the application is finally run with the commands shown in Box 8.16.

■ Box 8.16: Running a Django application

```
#cd into the project root
```

```
$cd weatherStationProject

#Sync the database
$ python manage.py syncdb

#Run the application
$ python manage.py runserver
```

8.5 Designing a RESTful Web API

In this section you will learn how to develop a RESTful web API. The example in this section uses the Django REST framework [89] introduced earlier for building a REST API. With the Django framework already installed, the Django REST framework can be installed as follows:

```
■ pip install djangorestframework
pip install markdown
pip install django-filter
```

After installing the Django REST framework, let us create a new Django project named *restfulapi*, and then start a new app called *myapp*, as follows:

```
■ django-admin.py startproject restfulapi
cd restfulapi
python manage.py startapp myapp
```

The REST API described in this section allows you to create, view, update and delete a collection of resources where each resource represents a sensor data reading from a weather monitoring station. Box 8.17 shows the Django model for such a station. The station model contains four fields - station name, timestamp, temperature, latitude and longitude. Box 8.18 shows the Django views for the REST API. ViewSets are used for the views that allow you to combine the logic for a set of related views in a single class.

Box 8.19 shows the serializers for the REST API. Serializers allow complex data (such as querysets and model instances) to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. Serializers also provide de-serialization, allowing parsed data to be converted back into complex types, after first validating the incoming data.

Box 8.20 shows the URL patterns for the REST API. Since ViewSets are used instead of views, we can automatically generate the URL conf for our API, by simply registering the viewsets with a router class. Routers automatically determining how the URLs for an application should be mapped to the logic that deals with handling incoming requests. Box 8.21 shows the settings for the REST API Django project.

■ Box 8.17: Django model for Weather Station - models.py

```
from django.db import models

class Station(models.Model):
   name = models.CharField(max_length=10)
   timestamp = models.CharField(max_length=10)
   temperature = models.CharField(max_length=5)
   lat = models.CharField(max_length=10)
   lon = models.CharField(max_length=10)
```

■ Box 8.18: Django views for Weather Station REST API - views.py

```
from myapp.models import Station
from rest_framework import viewsets
from django.shortcuts import render_to_response
from django.template import RequestContext
from myapp.serializers import StationSerializer
import requests
import json

class StationViewSet(viewsets.ModelViewSet):
   queryset = Station.objects.all()
   serializer_class = StationSerializer

def home(request):
   r=requests.get('http://127.0.0.1:8000/station/',
   auth=('username', 'password'))
   result=r.text
   output = json.loads(result)
   count=output['count']
   count=int(count)-1

   name=output['results'][count]['name']
```

```
    temperature=output['results'][count]['temperature']
    lat=output['results'][count]['lat']
    lon=output['results'][count]['lon']

    return render_to_response('index.html','name':name,
    'temperature':temperature, 'lat':  lat, 'lon':  lon,
    context_instance=RequestContext(request))
```

■ Box 8.19: Serializers for Weather Station REST API - serializers.py

```
from myapp.models import Station
from rest_framework import serializers

class StationSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Station
        fields = ('url', 'name','timestamp','timestamp',
        'temperature','lat','lon')
```

■ Box 8.20: Django URL patterns example - urls.py

```
from django.conf.urls import patterns, include, url
from django.contrib import admin
from rest_framework import routers
from myapp import views

admin.autodiscover()

router = routers.DefaultRouter()
router.register(r'station', views.StationViewSet)

urlpatterns = patterns('',
url(r'^', include(router.urls)),
url(r'^api-auth/', include('rest_framework.urls',
namespace='rest_framework')),
url(r'^admin/', include(admin.site.urls)),
url(r'^home/', 'myapp.views.home'),
)
```

■ Box 8.21: Django project settings example - settings.py

```
DATABASES = {
'default': {
'ENGINE': 'django.db.backends.mysql',
'NAME': 'weatherstation',
'USER': 'root',
'PASSWORD': 'password',
'HOST': '',
'PORT': '',
}
}
REST_FRAMEWORK = {
   'DEFAULT_PERMISSION_CLASSES':
   ('rest_framework.permissions.IsAdminUser',),
   'PAGINATE_BY': 10
}

INSTALLED_APPS = (
   'django.contrib.admin',
   'django.contrib.auth',
   'django.contrib.contenttypes',
   'django.contrib.sessions',
   'django.contrib.messages',
   'django.contrib.staticfiles',
   'myapp',
   'rest_framework',
)
```

After creating the Station REST API source files, the next step is to setup the database and then run the Django development web server as follows:

```
■ python manage.py syncdb
python manage.py runserver
```

■ Box 8.22: Using the Station REST API - CURL examples

```
#-----POST Example-------
$ curl -i -H "Content-Type:  application/json" -H
"Accept:application/json; indent=4" -X POST -d
```

```
'"name": "CityName", "timestamp": "1393926310",
"temperature": "28", "lat": "30.733315", "lon":
"76.779418"' -u arshdeep http://127.0.0.1:8000/station/
Enter host password for user 'arshdeep':
HTTP/1.0 201 CREATED
Date: Tue, 04 Mar 2014 11:21:29 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Type: application/json; indent=4
Location: http://127.0.0.1:8000/station/4/
Allow: GET, POST, HEAD, OPTIONS


   "url": "http://127.0.0.1:8000/station/4/",
   "name": "CityName",
   "timestamp": "1393926310",
   "temperature": "28",
   "lat": "30.733315",
   "lon": "76.779418"


#-----GET Examples-------
$ curl -i -H "Accept: application/json; indent=4" -u arsheep
http://127.0.0.1:8000/station/
Enter host password for user 'arshdeep':
HTTP/1.0 200 OK
Date: Tue, 04 Mar 2014 11:21:56 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Type: application/json; indent=4
Allow: GET, POST, HEAD, OPTIONS


   "count": 2,
   "next": null,
   "previous": null,
   "results": [

        "url": "http://127.0.0.1:8000/station/1/",
        "name": "CityName",
        "timestamp": "1393926457",
        "temperature": "20",
        "lat": "30.733315",
        "lon": "76.779418"
    ,
```

```
        "url":  "http://127.0.0.1:8000/station/2/",
        "name":  "CityName",
        "timestamp":  "1393926310",
        "temperature":  "28",
        "lat":  "30.733315",
        "lon":  "76.779418"

    ]


$ curl -i -H "Accept:  application/json; indent=4" -u arsheep
http://127.0.0.1:8000/station/1/
Enter host password for user 'arshdeep':
HTTP/1.0 200 OK
Date:  Tue, 04 Mar 2014 11:23:08 GMT
Server:  WSGIServer/0.1 Python/2.7.3
Vary:  Accept, Cookie
Content-Type:  application/json; indent=4
Allow:  GET, PUT, PATCH, DELETE, HEAD, OPTIONS


    "url":  "http://127.0.0.1:8000/station/1/",
    "name":  "CityName",
    "timestamp":  "1393926457",
    "temperature":  "20",
    "lat":  "30.733315",
    "lon":  "76.779418"


#-----PUT Example-------
$ curl -i -H "Content-Type:  application/json" -H
"Accept:  pplication/json; indent=4" -X PUT -d
'"name":  "CityName", "timestamp":  "1393926310",
"temperature":  "29", "lat":  "30.733315",
"lon":  "76.779418"' -u arshdeep
http://127.0.0.1:8000/station/1/
Enter host password for user 'arshdeep':
HTTP/1.0 200 OK
Date:  Tue, 04 Mar 2014 11:24:14 GMT
Server:  WSGIServer/0.1 Python/2.7.3
Vary:  Accept, Cookie
Content-Type:  application/json
Allow:  GET, PUT, PATCH, DELETE, HEAD, OPTIONS
```

```
"url": "http://127.0.0.1:8000/station/1/",
"name": "CityName", "timestamp": "1393926310",
"temperature": "29", "lat": "30.733315", "lon": "76.779418"

#-----DELETE Example-------
$curl -i -X DELETE -H "Accept: application/json; indent=4" -u arshdeep
http://127.0.0.1:8000/station/2/
Enter host password for user 'arshdeep':
HTTP/1.0 204 NO CONTENT
Date: Tue, 04 Mar 2014 11:24:55 GMT
Server: WSGIServer/0.1 Python/2.7.3
Vary: Accept, Cookie
Content-Length: 0
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
```

Box 8.22 shows examples of interacting with the Station REST API using CURL. The HTTP POST method is used to create a new resource, GET method is used to obtain information about a resource, PUT method is used to update a resource and DELETE method is used to delete a resource. Figure 8.11 shows the screenshots from the web browsable Station REST API.

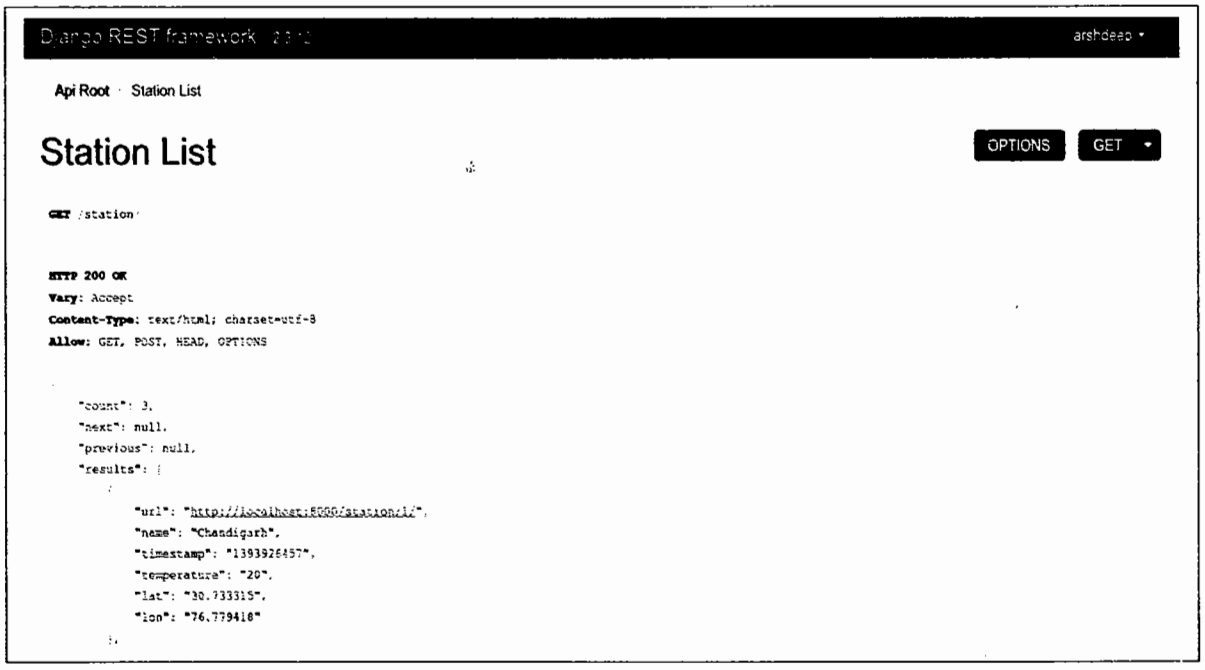

Figure 8.11: Screenshot from the web browsable Station REST API

8.6 Amazon Web Services for IoT

In this section you will learn how to use Amazon Web Services for IoT.

8.6.1 Amazon EC2

Amazon EC2 is an Infrastructure-as-a-Service (IaaS) provided by Amazon. EC2 delivers scalable, pay-as-you-go compute capacity in the cloud. EC2 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 purposes for IoT systems. For example, IoT developers can deploy IoT applications (developed in frameworks such as Django) on EC2, setup IoT platforms with REST web services, etc.

Let us look at some examples of using EC2. Box 8.23 shows the Python code for launching an EC2 instance. In this example, a connection to EC2 service is first established by calling *boto.ec2.connect_to_region*. The EC2 region, AWS access key and AWS secret key are passed to this function. After connecting to EC2 , a new instance is launched using the *conn.run_instances* function. The AMI-ID, instance type, EC2 key handle and security group are passed to this function. This function returns a reservation. The instances associated with the reservation are obtained using *reservation.instances*. Finally the status of an instance associated with a reservation is obtained using the *instance.update* function. In the example shown in Box 8.23, the program waits till the status of the newly launched instance becomes *running* and then prints the instance details such as public DNS, instance IP, and launch time.

■ Box 8.23: Python program for launching an EC2 instance

```
import boto.ec2
from time import sleep

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)
```

Box 8.24 shows the Python code for stopping an EC2 instance. In this example the *conn.get_all_instances* function is called to get information on all running instances. This function returns reservations. Next, the IDs of instances associated with each reservation are obtained. The instances are stopped by calling *conn.stop_instances* function to which the IDs of the instances to stop are passed.

■ Box 8.24: Python program for 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)
```

8.6.2 Amazon AutoScaling

Amazon AutoScaling allows automatically scaling Amazon EC2 capacity up or down according to user defined conditions. Therefore, with 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 and scale down capacity when the workload is low to save costs. AutoScaling can be used for auto scaling IoT applications and IoT platforms deployed on Amazon EC2.

Let us now look at some examples of using AutoScaling. Box 8.25 shows the Python code for creating an AutoScaling group. In this example, a connection to AutoScaling service is first established by calling *boto.ec2.autoscale.connect_to_region* function.

The EC2 region, AWS access key and AWS secret key are passed to this function.

After connecting to AutoScaling service, a new launch configuration is created by calling *conn.create_launch_configuration*. Launch configuration contains instructions on how to launch new instances including the AMI-ID, instance type, security groups, etc. After creating a launch configuration, it is then associated with a new AutoScaling group. AutoScaling group is created by calling
conn.create_auto_scaling_group. The settings for AutoScaling group include maximum and minimum number of instances in the group, launch configuration, availability zones, optional load balancer to use with the group, etc. After creating an AutoScaling group, the policies for scaling up and scaling down are defined. In this example, a scale up policy with adjustment type *ChangeInCapacity* and $scaling\_adjustment = 1$ is defined. Similarly a scale down policy with adjustment type *ChangeInCapacity* and $scaling\_adjustment = -1$ is defined. With the scaling policies defined, the next step is to create Amazon CloudWatch alarms that trigger these policies. In this example, alarms for scaling up and scaling down are created. The scale up alarm is defined using the *CPUUtilization* metric with the *Average* statistic and threshold greater 70% for a period of 60 sec. The scale up policy created previously is associated with this alarm. This alarm is triggered when the average CPU utilization of the instances in the group becomes greater than 70% for more than 60 seconds. The scale down alarm is defined in a similar manner with a threshold less than 50%.

Box 8.25: Python program for creating an AutoScaling group

```
import boto.ec2.autoscale
from boto.ec2.autoscale import LaunchConfiguration
from boto.ec2.autoscale import AutoScalingGroup
from boto.ec2.cloudwatch import MetricAlarm
from boto.ec2.autoscale import ScalingPolicy
import boto.ec2.cloudwatch

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 Autoscaling Service"

conn = boto.ec2.autoscale.connect_to_region(REGION,
```

```
      aws_access_key_id=ACCESS_KEY,
      aws_secret_access_key=SECRET_KEY)


print "Creating launch configuration"


lc = LaunchConfiguration(name='My-Launch-Config-2',
        image_id=AMI_ID,
        key_name=EC2_KEY_HANDLE,
        instance_type=INSTANCE_TYPE,
        security_groups = [ SECGROUP_HANDLE, ])


conn.create_launch_configuration(lc)

print "Creating auto-scaling group"

ag = AutoScalingGroup(group_name='My-Group',
        availability_zones=['us-east-1b'],
        launch_config=lc, min_size=1, max_size=2,
        connection=conn)

conn.create_auto_scaling_group(ag)

print "Creating auto-scaling policies"

scale_up_policy = ScalingPolicy(name='scale_up',
            adjustment_type='ChangeInCapacity',
           as_name='My-Group',
            scaling_adjustment=1,
           cooldown=180)

scale_down_policy = ScalingPolicy(name='scale_down',
            adjustment_type='ChangeInCapacity',
           as_name='My-Group',
            scaling_adjustment=-1,
           cooldown=180)


conn.create_scaling_policy(scale_up_policy)
conn.create_scaling_policy(scale_down_policy)


scale_up_policy = conn.get_all_policies( as_group='My-Group',
```

```
policy_names=['scale_up'])[0]
scale_down_policy = conn.get_all_policies( as_group='My-Group',
policy_names=['scale_down'])[0]

print "Connecting to CloudWatch"

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

alarm_dimensions = "AutoScalingGroupName":  'My-Group'

print "Creating scale-up alarm"

scale_up_alarm = MetricAlarm(
     name='scale_up_on_cpu', namespace='AWS/EC2',
     metric='CPUUtilization', statistic='Average',
     comparison='>', threshold='70',
     period='60', evaluation_periods=2,
     alarm_actions=[scale_up_policy.policy_arn] ,
     dimensions=alarm_dimensions)

cloudwatch.create_alarm(scale_up_alarm)

print "Creating scale-down alarm"

scale_down_alarm = MetricAlarm(
     name='scale_down_on_cpu', namespace='AWS/EC2',
     metric='CPUUtilization', statistic='Average',
     comparison='<', threshold='50',
     period='60', evaluation_periods=2,
     alarm_actions=[scale_down_policy.policy_arn],
     dimensions=alarm_dimensions)

cloudwatch.create_alarm(scale_down_alarm)
print "Done!"
```

8.6.3 Amazon S3

Amazon S3 is an online cloud-based data storage infrastructure for storing and retrieving a very large amount of data. S3 provides highly reliable, scalable, fast, fully redundant and affordable storage infrastructure. S3 can serve as a raw datastore (or "Thing Tank") for IoT systems for storing raw data, such as sensor data, log data, image, audio and video data.

Let us look at some examples of using S3. Box 8.26 shows the Python code for uploading

a file to Amazon S3 cloud storage. In this example, a connection to S3 service is first established by calling *boto.connect_s3* function. The AWS access key and AWS secret key are passed to this function. This example defines two functions *upload_to_s3_bucket_path* and *upload_to_s3_bucket_root*. The *upload_to_s3_bucket_path* function uploads the file to the S3 bucket specified at the specified path. The *upload_to_s3_bucket_root* function uploads the file to the S3 bucket root.

■ Box 8.26: 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')
```

8.6.4 Amazon RDS

Amazon RDS is a web service that allows you to create instances of MySQL, Oracle or Microsoft SQL Server in the cloud. With RDS, developers can easily set up, operate, and scale a relational database in the cloud.

RDS can serve as a scalable datastore for IoT systems. With RDS, IoT system developers can store any amount of data in scalable relational databases. Let us look at some examples of using RDS. Box 8.27 shows the Python code for launching an Amazon RDS instance. In this example, a connection to RDS service is first established by calling *boto.rds.connect_to_region*

function. The RDS region, AWS access key and AWS secret key are passed to this function. After connecting to RDS service, the *conn.create_dbinstance* function is called to launch a new RDS instance. The input parameters to this function include the instance ID, database size, instance type, database username, database password, database port, database engine (e.g. MySQL5.1), database name, security groups, etc. The program shown in Box 8.27 waits till the status of the RDS instance becomes *available* and then prints the instance details such as instance ID, create time, or instance end point.

■ Box 8.27: Python program for launching an RDS instance

```
import boto.rds
from time import sleep

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

REGION="us-east-1"
INSTANCE_TYPE="db.t1.micro"
ID = "MySQL-db-instance"
USERNAME = 'root'
PASSWORD = 'password'
DB_PORT = 3306
DB_SIZE = 5
DB_ENGINE = 'MySQL5.1'
DB_NAME = 'mytestdb'
SECGROUP_HANDLE="default"

print "Connecting to RDS"

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

print "Creating an RDS instance"

db = conn.create_dbinstance(ID, DB_SIZE, INSTANCE_TYPE,
USERNAME, PASSWORD, port=DB_PORT, engine=DB_ENGINE,
db_name=DB_NAME, security_groups = [
SECGROUP_HANDLE, ] )
print db

print "Waiting for instance to be up and running"
```

```
status = db.status
while not status == 'available':
   sleep(10)
   status = db.status

if status == 'available':
   print " \n RDS Instance is now running.  Instance details are:"
   print "Intance ID: " + str(db.id)
   print "Intance State:  " + str(db.status)
   print "Intance Create Time:  " + str(db.create_time)
   print "Engine:  " + str(db.engine)
   print "Allocated Storage:  " + str(db.allocated_storage)
   print "Endpoint:  " + str(db.endpoint)
```

Box 8.28 shows the Python code for creating a MySQL table, writing and reading from the table. This example uses the MySQLdb Python package. To connect to the MySQL RDS instance, the *MySQLdb.connect* function is called and the end point of the RDS instance, database username, password and port are passed to this function. After the connection to the RDS instance is established, a cursor to the database is obtained by calling *conn.cursor*. Next, a new database table named *TemperatureData* is created with *Id* as primary key and other columns. After creating the table some values are inserted. To execute the SQL commands for database manipulation, the commands are passed to the *cursor.execute* function.

■ Box 8.28: Python program for creating a MySQL table, writing and reading from the table

```
import MySQLdb

USERNAME = 'root'
PASSWORD = 'password'
DB_NAME = 'mytestdb'

print "Connecting to RDS instance"

conn = MySQLdb.connect (host =
"mysql-db-instance-3.c35qdifuf9ko.us-east-1.rds.amazonaws.com",
user = USERNAME,
passwd = PASSWORD,
db = DB_NAME,
port = 3306)

print "Connected to RDS instance"
```

```
cursor = conn.cursor ()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone ()
print "server version:", row[0]

cursor.execute ("CREATE TABLE TemperatureData(Id INT PRIMARY KEY,
Timestamp TEXT, Data TEXT) ")
cursor.execute ("INSERT INTO TemperatureData VALUES(1,
'1393926310', '20')")
cursor.execute ("INSERT INTO TemperatureData VALUES(2,
'1393926457', '25')")

cursor.execute("SELECT * FROM TemperatureData")
rows = cursor.fetchall()

for row in rows:
   print row

cursor.close ()
conn.close ()
```

8.6.5 Amazon DynamoDB

Amazon DynamoDB is a fully-managed, scalable, high performance No-SQL database service. DynamoDB can serve as a scalable datastore for IoT systems. With DynamoDB, IoT system developers can store any amount of data and serve any level of requests for the data.

Let us look at some examples of using DynamoDB. Box 8.29 shows the Python code for creating a DynamoDB table. In this example, a connection to DynamoDB service is first established by calling *boto.dynamodb.connect_to_region*. The DynamoDB region, AWS access key and AWS secret key are passed to this function. After connecting to DynamoDB service, a schema for the new table is created by calling *conn.create_schema*. The schema includes the hash key and range key names and types. A DynamoDB table is then created by calling *conn.create_table* function with the table schema, read units and write units as input parameters.

■ Box 8.29: Python program for creating a DynamoDB table

```
import boto.dynamodb
import time
```

```
from datetime import date

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to DynamoDB"

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

table_schema = conn.create_schema(
   hash_key_name='msgid',
   hash_key_proto_value=str,
   range_key_name='date',
   range_key_proto_value=str
   )

print "Creating table with schema:"
print table_schema


table = conn.create_table(
   name='my-test-table',
   schema=table_schema,
   read_units=1,
   write_units=1
   )

print "Creating table:"
print table

print "Done!"
```

Box 8.30 shows the Python code for writing and reading from a DynamoDB table. After establishing a connection with DynamoDB service, the *conn.get_table* is called to retrieve an existing table. The data written in this example consists of a JSON message with keys - *Body*, *CreatedBy* and *Time*. After creating the JSON message, a new DynamoDB table item is created by calling *table.new_item* and the hash key and range key is specified. The data item is finally committed to DynamoDB by calling *item.put*. To read data from DynamoDB, the *table.get_item* function is used with the hash key and range key as input parameters.

■ Box 8.30: Python program for writing and reading from a DynamoDB table

```
import boto.dynamodb
import time
from datetime import date

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to DynamoDB"

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

print "Listing available tables"
tables_list = conn.list_tables()
print tables_list

print "my-test-table description"
desc = conn.describe_table('my-test-table')
print desc

msg_datetime = time.asctime(time.localtime(time.time()))

print "Writing data"

table = conn.get_table("my-test-table")

hash_attribute = "Entry/" + str(date.today())

item_data =
   'Body': 'Test message',
   'CreatedBy': 'Vijay',
   'Time': msg_datetime,


item = table.new_item(
   hash_key=hash_attribute,
   range_key=str(date.today()),
   attrs=item_data
)
item.put()
```

```
print "Reading data"

table = conn.get_table('my-test-table')

read_data = table.get_item(
     hash_key=hash_attribute,
     range_key=str(date.today())
     )

print read_data
print "Done!"
```

8.6.6 Amazon Kinesis

Amazon Kinesis is a fully managed commercial service that allows real-time processing of streaming data. Kinesis scales automatically to handle high volume streaming data coming from large number of sources. The streaming data collected by Kinesis can be processed by applications running on Amazon EC2 instances or any other compute instance that can connect to Kinesis. Kinesis is well suited for IoT systems that generate massive scale data and have strict real-time requirements for processing the data. Kinesis allows rapid and continuous data intake and support data blobs of size upto 50Kb. The data producers (e.g. IoT devices) write data records to Kinesis streams. A data record comprises of a sequence number, a partition key and the data blob. Data records in a Kinesis stream are distributed in shards. Each shard provides a fixed unit of capacity and a stream can have multiple shards. A single shard of throughput allows capturing 1MB per second of data, at up to 1,000 PUT transactions per second and allows applications to read data at up to 2 MB per second.

Box 8.31 shows a Python program for writing to a Kinesis stream. This example follows a similar structure as the controller program in Box 8.4 that sends temperature data from an IoT device to the cloud. In this example a connection to the Kinesis service is first established and then a new Kinesis stream is either created (if not existing) or described. The data is written to the Kinesis stream using the *kinesis.put_record* function.

■ Box 8.31: Python program for writing to a Kinesis stream

```
import json
import time
import datetime
import boto.kinesis
from boto.kinesis.exceptions import ResourceNotFoundException
```

```
from random import randint

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

kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
streamName = "temperature"
partitionKey = "IoTExample"
shardCount = 1
global stream


def readTempSensor():
#Return random value
   return randint(20,30)

#Controller main function
def runController():
   temperature = readTempSensor()
   timestamp = datetime.datetime.utcnow()
   record=str(timestamp)+":"+str(temperature)
   print "Putting record in stream:  " + record
   response = kinesis.put_record( stream_name=streamName,
data=record, partition_key=partitionKey)
   print ("-= put seqNum:", response['SequenceNumber'])

def get_or_create_stream(stream_name, shard_count):
   stream = None
   try:
      stream = kinesis.describe_stream(stream_name)
      print (json.dumps(stream, sort_keys=True, indent=2, separators=(',',
':  ')))
   except ResourceNotFoundException as rnfe:
      while (stream is None) or (stream['StreamStatus'] is not 'ACTIVE'):
         print ('Could not find ACTIVE stream:0 trying to create.'.format(
            stream_name))
         stream = kinesis.create_stream(stream_name, shard_count)
         time.sleep(0.5)

   return stream


def setupController():
   global stream
```

```
    stream = get_or_create_stream(streamName, shardCount)

setupController()
while True:
    runController()
    time.sleep(1)
```

Box 8.32 shows a Python program for reading from a Kinesis stream. In this example a shard iterator is obtained using the *kinesis.get_shard_iterator* function. The shard iterator specifies the position in the shard from which you want to start reading data records sequentially. The data is read using the *kinesis.get_records* function which returns one or more data records from a shard.

■ Box 8.32: Python program for reading from a Kinesis stream

```
import json
import time
import boto.kinesis
from boto.kinesis.exceptions import ResourceNotFoundException
from boto.kinesis.exceptions import
ProvisionedThroughputExceededException
from boto.kinesis.exceptions import
ProvisionedThroughputExceededException

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

kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
streamName = "temperature"
partitionKey = "IoTExample"
shardCount = 1
iterator_type='LATEST'

stream = kinesis.describe_stream(streamName)
print (json.dumps(stream, sort_keys=True, indent=2,
separators=(',', ':  ')))
shards = stream['StreamDescription']['Shards']
print ('# Shard Count:', len(shards))

def processRecords(records):
    for record in records:
        text = record['Data'].lower()
```

```
        print 'Processing record with data:  '  + text

i=0
response = kinesis.get_shard_iterator(streamName, shards[0]['ShardId'],
'TRIM_HORIZON', starting_sequence_number=None)
next_iterator = response['ShardIterator']
print ('Getting next records using iterator:  ', next_iterator)
while i<10:
   try:
      response = kinesis.get_records(next_iterator, limit=1)
      #print response
      if len(response['Records']) > 0:
         #print 'Number of records fetched:'  +
str(len(response['Records']))
         processRecords(response['Records'] )

      next_iterator = response['NextShardIterator']
      time.sleep(1)
      i=i+1

   except ProvisionedThroughputExceededException as ptee:
      print (ptee.message)
      time.sleep(5)
```

8.6.7 Amazon SQS

Amazon SQS offers a highly scalable and reliable hosted queue for storing messages as they travel between distinct components of applications. SQS guarantees only that messages arrive, not that they arrive in the same order in which they were put in the queue. Though, at first look, Amazon SQS may seem to be similar to Amazon Kinesis, however, both are intended for very different types of applications. While Kinesis is meant for real-time applications that involve high data ingress and egress rates, SQS is simply a queue system that stores and releases messages in a scalable manner.

SQS can be used in distributed IoT applications in which various application components need to exchange messages. Let us look at some examples of using SQS. Box 8.33 shows the Python code for creating an SQS queue. In this example, a connection to SQS service is first established by calling *boto.sqs.connect_to_region*. The AWS region, access key and secret key are passed to this function. After connecting to SQS service, *conn.create_queue* is called to create a new queue with queue name as input parameter. The function *conn.get_all_queues* is used to retrieve all SQS queues.

■ Box 8.33: Python program for creating an SQS queue

```
import boto.sqs

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to SQS"

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

queue_name = 'mytestqueue'

print "Creating queue with name:  " + queue_name
q = conn.create_queue(queue_name)

print "Created queue with name:  " + queue_name

print " \n Getting all queues"

rs = conn.get_all_queues()

for item in rs:
   print item
```

Box 8.34 shows the Python code for writing to an SQS queue. After connecting to an SQS queue, the *queue.write* is called with the message as input parameter.

■ Box 8.34: Python program for writing to an SQS queue

```
import boto.sqs
from boto.sqs.message import Message
import time

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

REGION="us-east-1"
```

```
print "Connecting to SQS"

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

queue_name = 'mytestqueue'

print "Connecting to queue:  " + queue_name
q = conn.get_all_queues(prefix=queue_name)

msg_datetime = time.asctime(time.localtime(time.time()))

msg = "Test message generated on:  " + msg_datetime
print "Writing to queue:  " + msg

m = Message()
m.set_body(msg)
status = q[0].write(m)

print "Message written to queue"

count = q[0].count()

print "Total messages in queue:  " + str(count)
```

Box 8.35 shows the Python code for reading from an SQS queue. After connecting to an SQS queue, the *queue.read* is called to read a message from a queue.

■ Box 8.35: Python program for reading from an SQS queue

```
import boto.sqs
from boto.sqs.message import Message

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

REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect_to_region(
     REGION,
```

```
      aws_access_key_id=ACCESS_KEY,
      aws_secret_access_key=SECRET_KEY)

queue_name = 'mytestqueue'

print "Connecting to queue:  " + queue_name
q = conn.get_all_queues(prefix=queue_name)

count = q[0].count()

print "Total messages in queue:  " + str(count)

print "Reading message from queue"

for i in range(count):
   m = q[0].read()
   print "Message %d:  %s" % (i+1,str(m.get_body()))
   q[0].delete_message(m)

print "Read %d messages from queue" % (count)
```

8.6.8 Amazon EMR

Amazon EMR is a web service that utilizes Hadoop framework running on Amazon EC2 and Amazon S3. EMR allows processing of massive scale data, hence, suitable for IoT applications that generate large volumes of data that needs to be analyzed. Data processing jobs are formulated with the MapReduce parallel data processing model.

MapReduce is a parallel data processing model for processing and analysis of massive scale data [85]. MapReduce model has two phases: Map and Reduce. MapReduce programs are written in a functional programming style to create Map and Reduce functions. The input data to the map and reduce phases is in the form of key-value pairs.

Consider an IoT system that collects data from a machine (or sensor data) which is logged in a cloud storage (such as Amazon S3) and analyzed on hourly basis to generate alerts if a certain sequence occurred more than a predefined number of times. Since the scale of data involved in such applications can be massive, MapReduce is an ideal choice for processing such data.

Let us look at a MapReduce example that finds the number of occurrences of a sequence from a log. Box 8.36 shows the Python code for launching an Elastic MapReduce job. In this example, a connection to EMR service is first established by calling *boto.emr.connect_to_region*. The AWS region, access key and secret key are passed to this function. After connecting to EMR service, a jobflow step is created. There are two types of steps - streaming and

custom jar. To create a streaming job an object of the *StreamingStep* class is created by specifying the job name, locations of the mapper, reducer, input and output. The job flow is then started using the *conn.run_jobflow* function with streaming step object as input. When the MapReduce job completes, the output can be obtained from the output location on the S3 bucket specified while creating the streaming step.

■ Box 8.36: Python program for launching an EMR job

```
import boto.emr
from boto.emr.step import StreamingStep
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to EMR"

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

print "Creating streaming step"

step = StreamingStep(name='Sequence Count',
     mapper='s3n://mybucket/seqCountMapper.py',
     reducer='s3n://mybucket/seqCountReducer.py',
     input='s3n://mybucket/data/',
     output='s3n://mybucket/seqcountoutput/')


print "Creating job flow"

jobid = conn.run_jobflow(name='Sequence Count Jobflow',
     log_uri='s3n://mybucket/wordcount_logs',
     steps=[step])

print "Submitted job flow"

print "Waiting for job flow to complete"

status = conn.describe_jobflow(jobid)
print status.state
```

```
while status.state != 'COMPLETED' or status.state != 'FAILED':
   sleep(10)
   status = conn.describe_jobflow(jobid)

print "Job status:  " + str(status.state)

print "Done!"
```

Box 8.37 shows the sequence count mapper program in Python. The mapper reads the data from standard input (stdin) and for each line in input in which the sequence occurs, the mapper emits a key-value pair where key is the sequence and value is equal to 1.

■ Box 8.37: Sequence count Mapper in Python

```
#!/usr/bin/env python
import sys

#Enter the sequence to search
seq='123'
for line in sys.stdin:
   line = line.strip()
   if seq in line:
      print '%st%s' % (seq, 1)
```

Box 8.38 shows the sequence count reducer program in Python. The key-value pairs emitted by the map phase are shuffled to the reducers and grouped by the key. The reducer reads the key-value pairs grouped by the same key from the standard input (stdin) and sums up the occurrences to compute the count for each sequence.

■ Box 8.38: Sequence count Reducer in Python

```
#!/usr/bin/env python
from operator import itemgetter
import sys

current_seq = None
current_count = 0
seq = None

for line in sys.stdin:
```

```
    line = line.strip()
    seq, count = line.split('t', 1)
    current_count += count

    try:
        count = int(count)
    except ValueError:
        continue

    if current_seq == seq:
        current_count += count
else:
    if current_seq:
        print '%st%s' % (current_seq,current_count)
    current_count = count
    current_seq = seq

if current_seq == seq:
    print '%st%s' % (current_seq, current_count)
```

8.7 SkyNet IoT Messaging Platform

SkyNet is an open source instant messaging platform for Internet of Things. The SkyNet API supports both HTTP REST and real-time WebSockets. SkyNet allows you to register devices (or nodes) on the network. A device can be anything including sensors, smart home devices, cloud resources, drones, etc. Each device is assigned a UUID and a secret token. Devices or client applications can subscribe to other devices and receive/send messages.

Box 8.39 shows the commands to setup SkyNet on a Linux machine. Box 8.40 shows a sample configuration for SkyNet. Box 8.41 shows examples of using SkyNet. The first step is to create a device on SkyNet. The POST request to create a device returns the UUID and token of the created device. The box also shows examples of updating a device, retrieving last 10 events related to a device, subscribing to a device and sending a message to a device. Box 8.42 shows the code for a Python client that performs various functions such as subscribing to a device, sending a message and retrieving the service status.

■ Box 8.39: Commands for Setting up SkyNet

```
#Install DB Redis:
sudo apt-get install redis-server

#Install MongoDB:
```

```
sudo apt-key adv -keyserver keyserver.ubuntu.com -recv 7F0CEB10
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen' |
sudo tee /etc/apt/sources.list.d/10gen.list
sudo apt-get update
sudo apt-get install mongodb-10gen

#Install dependencies
sudo apt-get install git
sudo apt-get install software-properties-common
sudo apt-get install npm

#Install Node.JS
sudo apt-get update
sudo apt-get install -y python-software-properties python g++ make
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install nodejs

#Install Skynet:
git clone https://github.com/skynetim/skynet.git
npm config set registry http://registry.npmjs.org/
npm install
```

■ Box 8.40: Sample SkyNet configuration file

```
module.exports = {
   databaseUrl:  "mongodb://localhost:27017/skynet",
   port:  3000,
   log:  true,
   rateLimit:  10, // 10 transactions per user per secend
   redisHost:  "127.0.0.1",
   redisPort:  "6379"
};
```

■ Box 8.41: Using the SkyNet REST API

```
#Creating a device
$curl -X POST -d "name=mydevicename&token=mytoken&color=green"
http://localhost:3000/devices
```

```
{"name":"mydevicename","token":"mytoken","ipAddress":"127.0.0.1",
"uuid":"myuuid","timestamp":1394181626324,"channel":"main",
"online":false,"_id":"531985fa16ac510d4c000006","eventCode":400}

--------------------
#Listing devices
$curl http://localhost:3000/devices/myuuid

{"name":"mydevicename","ipAddress":"127.0.0.1","uuid":"myuuid",
"timestamp":1394181626324, "channel":"main","online":false,
"_id":"531985fa16ac510d4c000006", "eventCode":500}

---------------------
#Update a device
$curl -X PUT -d "token=mytoken&color=red"
http://localhost:3000/devices/myuuid

#Get last 10 events for a device
$curl -X GET http://localhost:3000/events/myuuid?token=mytoken

{"events":[{"color":"red","fromUuid":"myuuid",
"timestamp":1394181722052, "eventCode":300,"id":"mytoken"},{
"name":"mydevicename","ipAddress":"127.0.0.1" ,
"uuid":"myuuid",
"timestamp":1394181626324, "channel":"main","online":false,
"eventCode":500, "id":"531985fa16ac510d4c000006"}]}

---------------------
#Subscribing to a device
$curl -X GET http://localhost:3000/subscribe/myuuid?token=mytoken

---------------------
#Sending a message
$curl -X POST -d '{"devices":  "myuuid", "message":
{"color":"red"}}' http://localhost:3000/messages
```

■ Box 8.42: Python client for SkyNet

```
from socketIO_client import SocketIO

HOST='<enter host IP>'
PORT=3000
UUID='<enter UUID>'
```

```
TOKEN='<enter Token>'

def on_status_response(*args):
   print 'Status:  ', args

def on_ready_response(*args):
   print 'Ready:  ', args

def on_noready_response(*args):
   print 'Not Ready:  ', args

def on_sub_response(*args):
   print 'Subscribed:  ', args

def on_msg_response(*args):
   print 'Message Received:  ', args

def on_whoami_response(*args):
   print 'Who Am I : ', args

def on_id_response(*args):
   print 'Websocket connecting to Skynet with
socket id:  '  + args[0]['socketid']
   print 'Sending arguments:  ', type(args), args
   socketIO.emit('identity', 'uuid':UUID, 'socketid':
args[0]['socketid'], 'token':TOKEN)

def on_connect(*args):
   print 'Requesting websocket connection to Skynet'
   socketIO.on('identify', on_id_response)
   socketIO.on('ready', on_ready_response)
   socketIO.on('notReady', on_noready_response)

socketIO = SocketIO(HOST, PORT)
socketIO.on('connect', on_connect)

socketIO.emit('status', on_status_response)
socketIO.emit('subscribe', 'uuid':UUID,'token':  TOKEN, on_sub_response)

socketIO.emit('whoami', 'uuid':UUID, on_whoami_response)

socketIO.emit('message', 'devices':  UUID, 'message':  'color':'purple')
socketIO.on('message', on_msg_response)
socketIO.wait()
```

Summary

In this chapter you learned about various cloud computing services and their applications for IoT. You learned about the WAMP protocol and the AutoBahn framework. WAMP is a sub-protocol of Websocket which provides publish-subscribe and RPC messaging patterns. You learned about the Xively Platform-as-a-Service that can be used for creating solutions for Internet of Things. Xively platform comprises of a message bus for real-time message management and routing, data services for time series archiving, directory services that provides a searchable directory of objects and business services for device provisioning and management. You learned how to send data to and retrieve data from Xively.

You learned about Django which is an open source web application framework for developing web applications in Python. Django is based on the Model-Template-View architecture. You also learned how to develop a Django application made up of model, view and templates. You learned how to develop a RESTful web API.

You learned about various commercial cloud services offered by Amazon. Amazon EC2 is a computing service from Amazon. You learned how to programmatically launch an Amazon EC2 instance. Amazon AutoScaling allows automatically scaling Amazon EC2 capacity up or down according to user defined conditions. You also learned how to programmatically create an AutoScaling group, define AutoScaling policies and CloudWatch alarms for triggering the AutoScaling policies. Amazon S3 is an online cloud-based data storage from Amazon. You learned how to programmatically upload a file to an S3 bucket. Amazon RDS is a cloud-based relational database service. You learned how to programmatically launch an RDS instance, view running instances, connect to an instance, create a MySQL table, write and read from the table on the RDS instance. Amazon DynamoDB is a No-SQL database service. You learned how to programmatically create a DynamoDB table, write and read from a DynamoDB table. Amazon SQS is a scalable queuing service from Amazon. You learned how to programmatically create an SQS queue, write messages to a queue and read messages from a queue. Amazon EMR is a MapReduce web service. You learned how to programmatically create an EMR job. Finally, you learned about the SkyNet messaging platform for IoT.

Review Questions

1. What is the difference between a Xively data stream and a channel?
2. Describe the architecture of a Django application.
3. What is the function of URL patterns in Django?
4. What is the purpose of an Amazon AutoScaling group? Describe the steps involved in creating an AutoScaling group.

5. What is Amazon DynamoDB? Describe an application that can benefit from Amazon DynamoDB.
6. Describe the use of Amazon Kinesis for IoT.
7. What are the uses of messaging queues? What are the message formats supported by Amazon SQS?
8. What does a MapReduce job comprise of?
9. What protocols does the SkyNet messaging platform support?