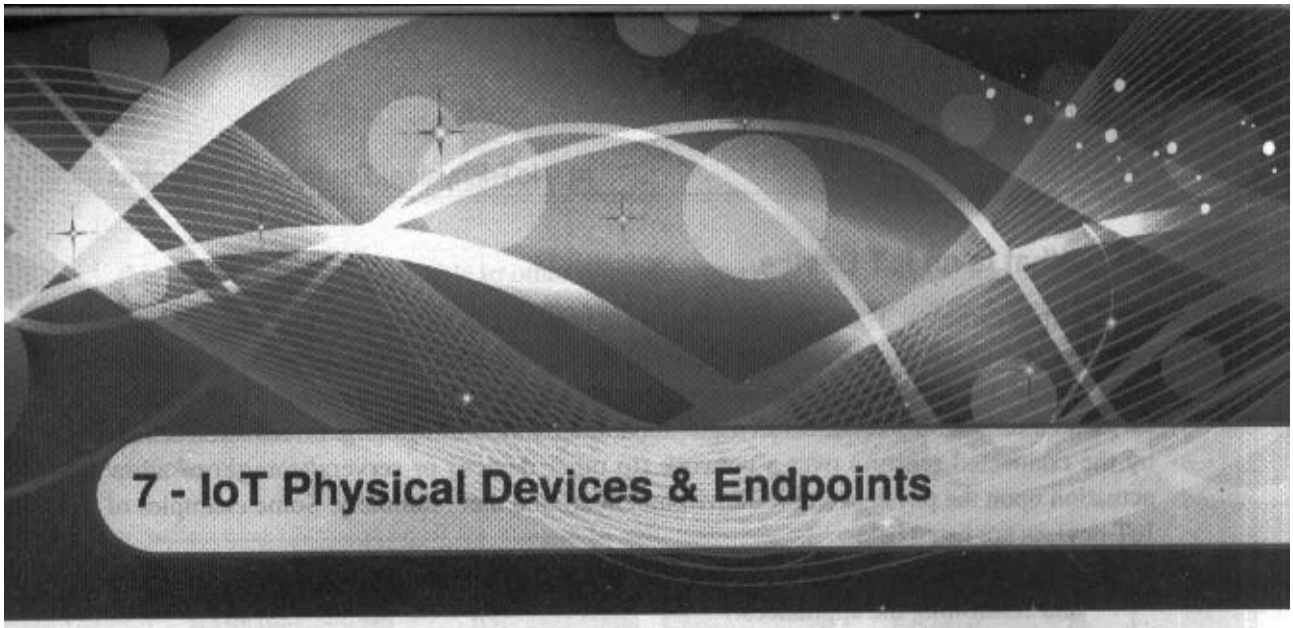


7 - IoT Physical Devices & Endpoints

This Chapter covers

- Basic building blocks of an IoT Device
- Exemplary Device: Raspberry Pi
- Raspberry Pi interfaces
- Programming Raspberry Pi with Python
- Other IoT devices

7.1 What is an IoT Device

As described earlier, a "Thing" in Internet of Things (IoT) can be any object that has a unique identifier and which can send/receive data (including user data) over a network (e.g., smart phone, smart TV, computer, refrigerator, car, etc.). IoT devices are connected to the Internet and send information about themselves or about their surroundings (e.g. information sensed by the connected sensors) over a network (to other devices or servers/storage) or allow actuation upon the physical entities/environment around them remotely. Some examples of IoT devices are listed below:

- A home automation device that allows remotely monitoring the status of appliances and controlling the appliances.
- An industrial machine which sends information abouts its operation and health monitoring data to a server.
- A car which sends information about its location to a cloud-based service.
- A wireless-enabled wearable device that measures data about a person such as the number of steps walked and sends the data to a cloud-based service.

7.1.1 Basic building blocks of an IoT Device

An IoT device can consist of a number of modules based on functional attributes, such as:

- Sensing: Sensors can be either on-board the IoT device or attached to the device. IoT device can collect various types of information from the on-board or attached sensors such as temperature, humidity, light intensity, etc. The sensed information can be communicated either to other devices or cloud-based servers/storage.
- Actuation: IoT devices can have various types of actuators attached that allow taking actions upon the physical entities in the vicinity of the device. For example, a relay switch connected to an IoT device can turn an appliance on/off based on the commands sent to the device.
- Communication: Communication modules are responsible for sending collected data to other devices or cloud-based servers/storage and receiving data from other devices and commands from remote applications.
- Analysis & Processing: Analysis and processing modules are responsible for making sense of the collected data.

The representative IoT device used for the examples in this book is the widely used single-board mini computer called Raspberry Pi (explained in later sections). The use of Raspberry Pi is intentional since these devices are widely accessible, inexpensive, and available from multiple vendors. Furthermore, extensive information is available on their programming and use both on the Internet and in other textbooks. The principles we teach in

this book are just as applicable to other (including proprietary) IoT endpoints, in addition to Raspberry Pi. Before we look at the specifics of Raspberry Pi, let us first look at the building blocks of a generic single-board computer (SBC) based IoT device.

Figure 7.1 shows a generic block diagram of a single-board computer (SBC) based IoT device that includes CPU, GPU, RAM, storage and various types of interfaces and peripherals.

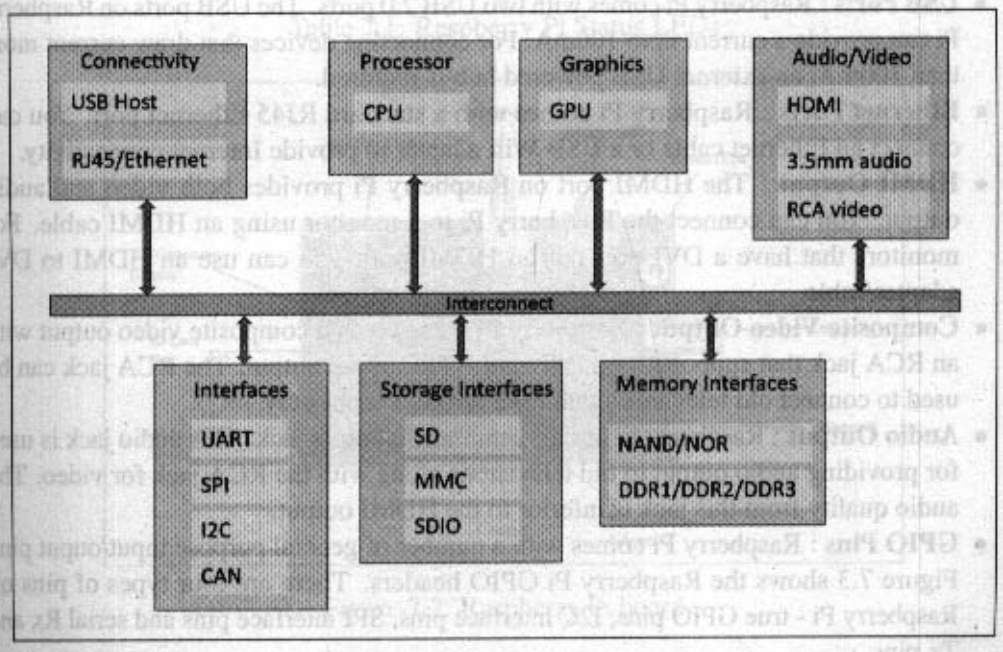


Figure 7.1: Block diagram of an IoT Device

7.2 Exemplary Device: Raspberry Pi

Raspberry Pi [104] is a low-cost mini-computer with the physical size of a credit card. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do. In addition to this, Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins. Since Raspberry Pi runs Linux operating system, it supports Python "out of the box".

7.3 About the Board

Figure 7.2 shows the Raspberry Pi board with the various components/peripherals labeled.

- **Processor & RAM** : Raspberry Pi is based on an ARM processor. The latest version of Raspberry Pi (Model B, Revision 2) comes with 700 MHz Low Power ARM1176JZ-F processor and 512 MB SDRAM.
- **USB Ports** : Raspberry Pi comes with two USB 2.0 ports. The USB ports on Raspberry Pi can provide a current upto 100mA. For connecting devices that draw current more than 100mA, an external USB powered hub is required.
- **Ethernet Ports** : Raspberry Pi comes with a standard RJ45 Ethernet port. You can connect an Ethernet cable or a USB Wifi adapter to provide Internet connectivity.
- **HDMI Output** : The HDMI port on Raspberry Pi provides both video and audio output. You can connect the Raspberry Pi to a monitor using an HDMI cable. For monitors that have a DVI port but no HDMI port, you can use an HDMI to DVI adapter/cable.
- **Composite Video Output** : Raspberry Pi comes with a composite video output with an RCA jack that supports both PAL and NTSC video output. The RCA jack can be used to connect old televisions that have an RCA input only.
- **Audio Output** : Raspberry Pi has a 3.5mm audio output jack. This audio jack is used for providing audio output to old televisions along with the RCA jack for video. The audio quality from this jack is inferior to the HDMI output.
- **GPIO Pins** : Raspberry Pi comes with a number of general purpose input/ouput pins. Figure 7.3 shows the Raspberry Pi GPIO headers. There are four types of pins on Raspberry Pi - true GPIO pins, I2C interface pins, SPI interface pins and serial Rx and Tx pins.
- **Display Serial Interface (DSI)** : The DSI interface can be used to connect an LCD panel to Raspberry Pi.
- **Camera Serial Interface (CSI)** : The CSI interface can be used to connect a camera module to Raspberry Pi.
- **Status LEDs** : Raspberry Pi has five status LEDs. Table 7.1 lists Raspberry Pi status LEDs and their functions.
- **SD Card Slot** : Raspberry Pi does not have a built in operating system and storage. You can plug-in an SD card loaded with a Linux image to the SD card slot. Appendix-A provides instructions on setting up New Out-of-the-Box Software (NOOBS) on Raspberry Pi. You will require atleast an 8GB SD card for setting up NOOBS.
- **Power Input** : Raspberry Pi has a micro-USB connector for power input.

Status LED	Function
ACT	SD card access
PWR	3.3V Power is present
FDX	Full duplex LAN connected
LNK	Link/Network activity
100	100 Mbit LAN connected

Table 7.1: Raspberry Pi Status LEDs

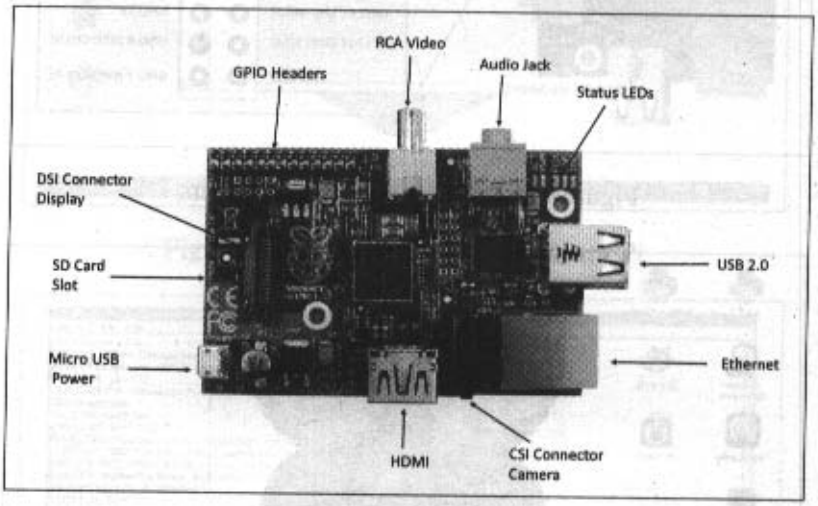


Figure 7.2: Raspberry Pi board

7.4 Linux on Raspberry Pi

Raspberry Pi supports various flavors of Linux including:

- **Raspbian** Raspbian Linux is a Debian Wheezy port optimized for Raspberry Pi. This is the recommended Linux for Raspberry Pi. Appendix-1 provides instructions on setting up Raspbian on Raspberry Pi.
- **Arch** : Arch is an Arch Linux port for AMD devices.
- **Pidora** : Pidora Linux is a Fedora Linux optimized for Raspberry Pi.
- **RaspBMC** : RaspBMC is an XBMC media-center distribution for Raspberry Pi.
- **OpenELEC** : OpenELEC is a fast and user-friendly XBMC media-center distribution.
- **RISC OS** : RISC OS is a very fast and compact operating system.

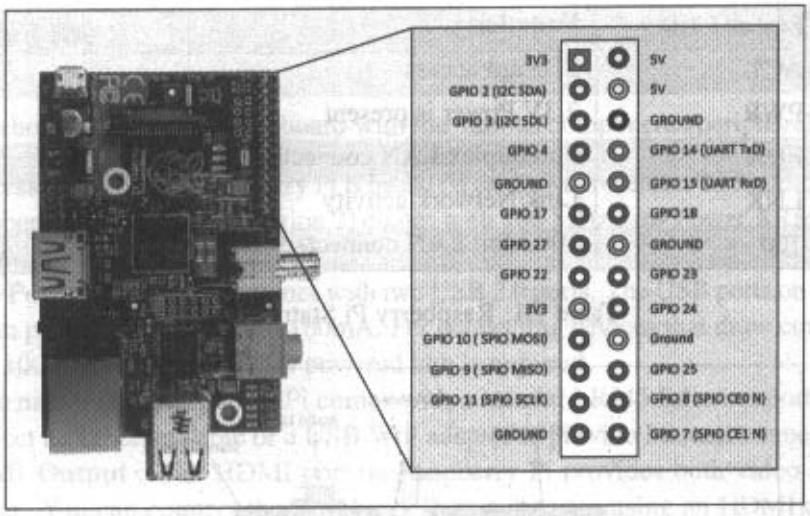


Figure 7.3: Raspberry Pi GPIO headers

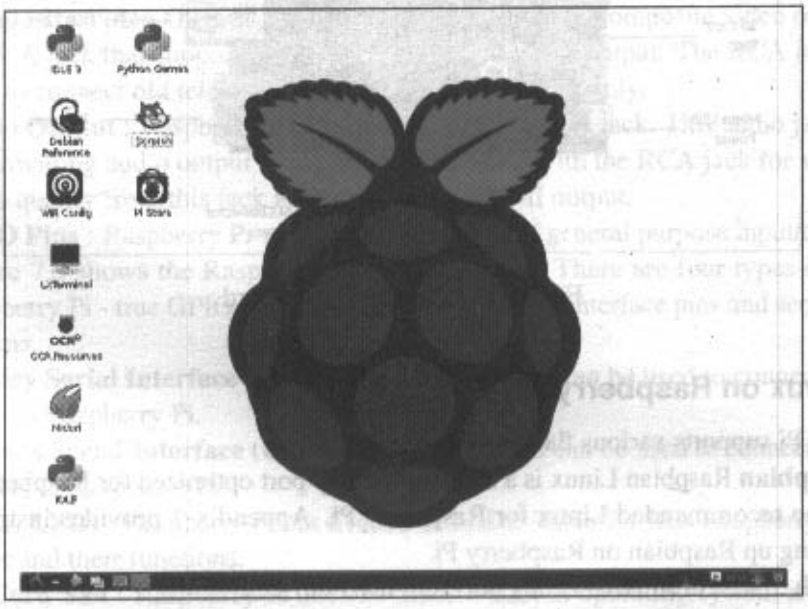


Figure 7.4: Rasbian Linux desktop

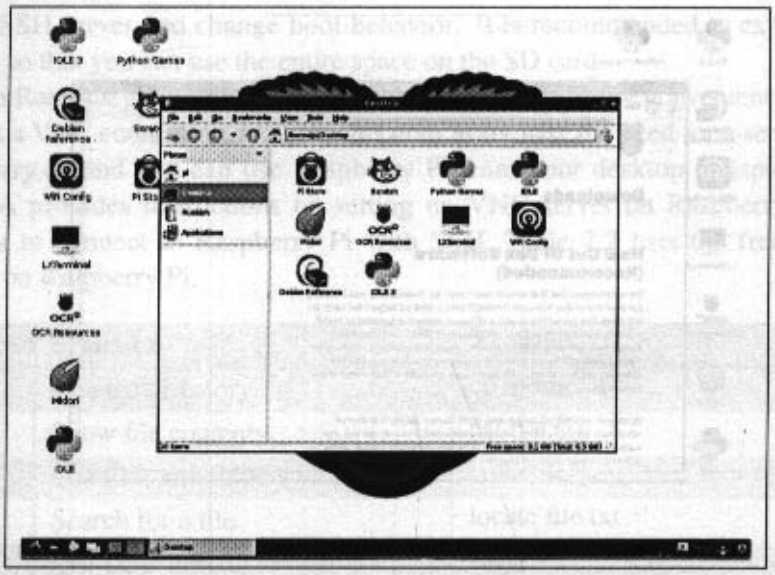


Figure 7.5: File explorer on Raspberry Pi

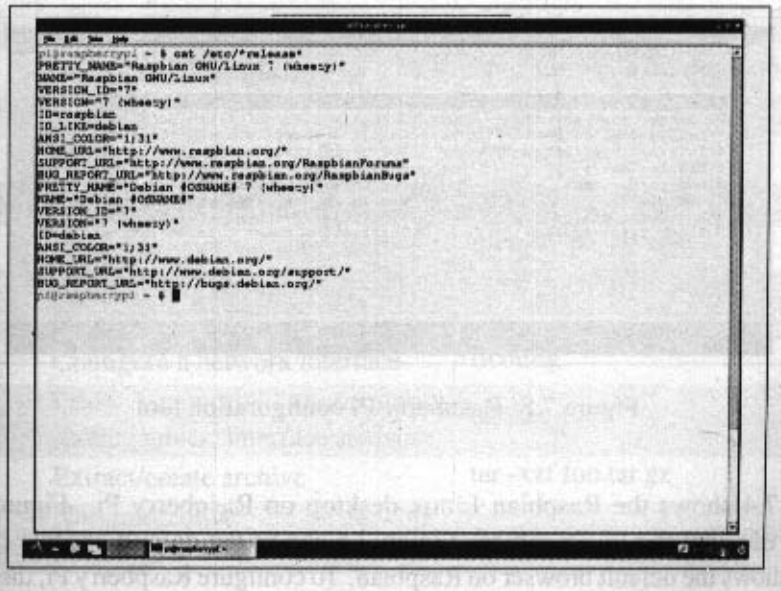


Figure 7.6: Console on Raspberry Pi

Figure 7.7: Browser on Raspberry Pi

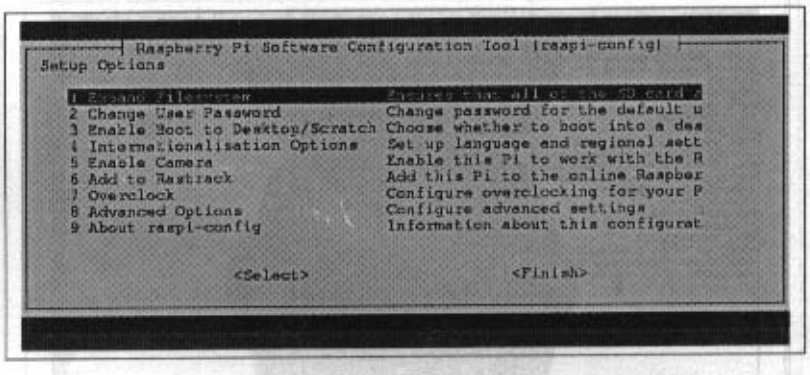


Figure 7.8: Raspberry Pi configuration tool

Figure 7.4 shows the Raspbian Linux desktop on Raspberry Pi. Figure 7.5 shows the default file explorer on Raspbian. Figure 7.6 shows the default console on Raspbian. Figure 7.7 shows the default browser on Raspbian. To configure Raspberry Pi, the raspi-config tool is used which can be launched from command line as (\$raspi-config) as shown in Figure 7.8. Using the configuration tool you can expand root partition to fill SD card, set keyboard layout, change password, set locale and timezone, change memory split, enable

or disable SSH server and change boot behavior. It is recommended to expand the root file-system so that you can use the entire space on the SD card.

Though Raspberry Pi comes with an HDMI output, it is more convenient to access the device with a VNC connection or SSH. This does away with the need for a separate display for Raspberry Pi and you can use Raspberry Pi from your desktop or laptop computer. Appendix-A provides instructions on setting up VNC server on Raspberry Pi and the instructions to connect to Raspberry Pi with SSH. Table 7.2 lists the frequently used commands on Raspberry Pi.

Command	Function	Example
cd	Change directory	cd /home/pi
cat	Show file contents	cat file.txt
ls	List files and folders	ls /home/pi
locate	Search for a file	locate file.txt
lsusb	List USB devices	lsusb
pwd	Print name of present working directory	pwd
mkdir	Make directory	mkdir /home/pi/new
mv	Move (rename) file	mv sourceFile.txt destinationFile.txt
rm	Remove file	rm file.txt
reboot	Reboot device	sudo reboot
shutdown	Shutdown device	sudo shutdown -h now
grep	Print lines matching a pattern	grep -r "pi" /home/
df	Report file system disk space usage	df -Th
ifconfig	Configure a network interface	ifconfig
netstat	Print network connections, routing tables, interface statistics	netstat -lntp
tar	Extract/create archive	tar -xzf foo.tar.gz
wget	Non-interactive network downloader	wget http://example.com/file.tar.gz

Table 7.2: Raspberry Pi frequently used commands

7.5 Raspberry Pi Interfaces

Raspberry Pi has serial, SPI and I2C interfaces for data transfer as shown in Figure 7.3.

7.5.1 Serial

The serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serial peripherals.

7.5.2 SPI

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheral devices. In an SPI connection, there is one master device and one or more peripheral devices. There are five pins on Raspberry Pi for SPI interface:

- **MISO (Master In Slave Out)** : Master line for sending data to the peripherals.
- **MOSI (Master Out Slave In)** : Slave line for sending data to the master.
- **SCK (Serial Clock)** : Clock generated by master to synchronize data transmission
- **CE0 (Chip Enable 0)** : To enable or disable devices.
- **CE0 (Chip Enable 1)** : To enable or disable devices.

7.5.3 I2C

The I2C interface pins on Raspberry Pi allow you to connect hardware modules. I2C interface allows synchronous data transfer with just two pins - SDA (data line) and SCL (clock line).

7.6 Programming Raspberry Pi with Python

In this section you will learn how to get started with developing Python programs on Raspberry Pi. Raspberry Pi runs Linux and supports Python out of the box. Therefore, you can run any Python program that runs on a normal computer. However, it is the general purpose input/output capability provided by the GPIO pins on Raspberry Pi that makes it useful device for Internet of Things. You can interface a wide variety of sensor and actuators with Raspberry Pi using the GPIO pins and the SPI, I2C and serial interfaces. Input from the sensors connected to Raspberry Pi can be processed and various actions can be taken, for instance, sending data to a server, sending an email, triggering a relay switch.

7.6.1 Controlling LED with Raspberry Pi

Let us start with a basic example of controlling an LED from Raspberry Pi. Figure 7.9 shows the schematic diagram of connecting an LED to Raspberry Pi. Box 7.1 shows how to turn

the LED on/off from command line. In this example the LED is connected to GPIO pin 18. You can connect the LED to any other GPIO pin as well.

Box 7.2 shows a Python program for blinking an LED connected to Raspberry Pi every second. The program uses the RPi. GPIO module to control the GPIO on Raspberry Pi. In this program we set pin 18 direction to output and then write *True/False* alternatively after a delay of one second.

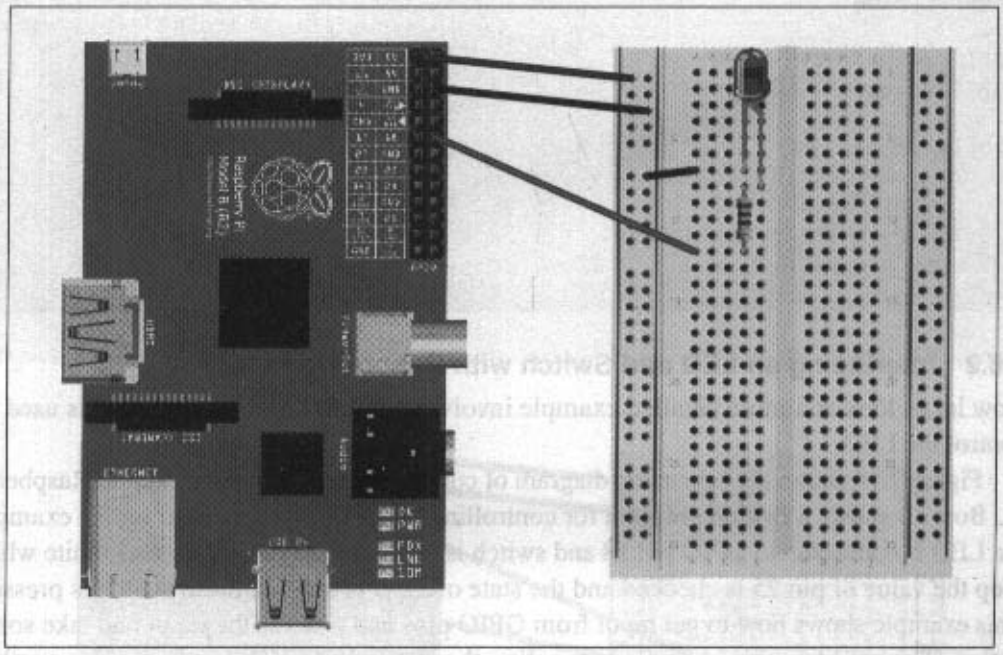


Figure 7.9: Controlling LED with Raspberry Pi

■ Box 7.1: Switching LED on/off from Raspberry Pi console

```
$echo 18 > /sys/class/gpio/export
$cd /sys/class/gpio/gpio18

#Set pin 18 direction to out
$echo out > direction

#Turn LED on
$echo 1 > value
```

```
#Turn LED off
$echo 0 > value
```

Box 7.2: Python program for blinking LED

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(18, GPIO.OUT)

while True:
   GPIO.output(18, True)
   time.sleep(1)
   GPIO.output(18, False)
   time.sleep(1)
```

7.6.2 Interfacing an LED and Switch with Raspberry Pi

Now let us look at a more detailed example involving an LED and a switch that is used to control the LED.

Figure 7.10 shows the schematic diagram of connecting an LED and switch to Raspberry Pi. Box 7.3 shows a Python program for controlling an LED with a switch. In this example the LED is connected to GPIO pin 18 and switch is connected to pin 25. In the infinite while loop the value of pin 25 is checked and the state of LED is toggled if the switch is pressed. This example shows how to get input from GPIO pins and process the input and take some action. The action in this example is toggling the state of an LED. Let us look at another example, in which the action is an email alert. Box 7.4 shows a Python program for sending an email on switch press. Note that the structure of this program is similar to the program in Box 7.3. This program uses the Python SMTP library for sending an email when the switch connected to Raspberry Pi is pressed.

Box 7.3: Python program for controlling an LED with a switch

```
from time import sleep
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
```

```
#Switch Pin
GPIO.setup(25, GPIO.IN)

#LED Pin
GPIO.setup(18, GPIO.OUT)

state=false

def toggleLED(pin):
   state = not state
   GPIO.output(pin, state)

while True:
   try:
      if (GPIO.input(25) == True):
         toggleLED(pin)
      sleep(.01)
   except KeyboardInterrupt:
      exit()
```

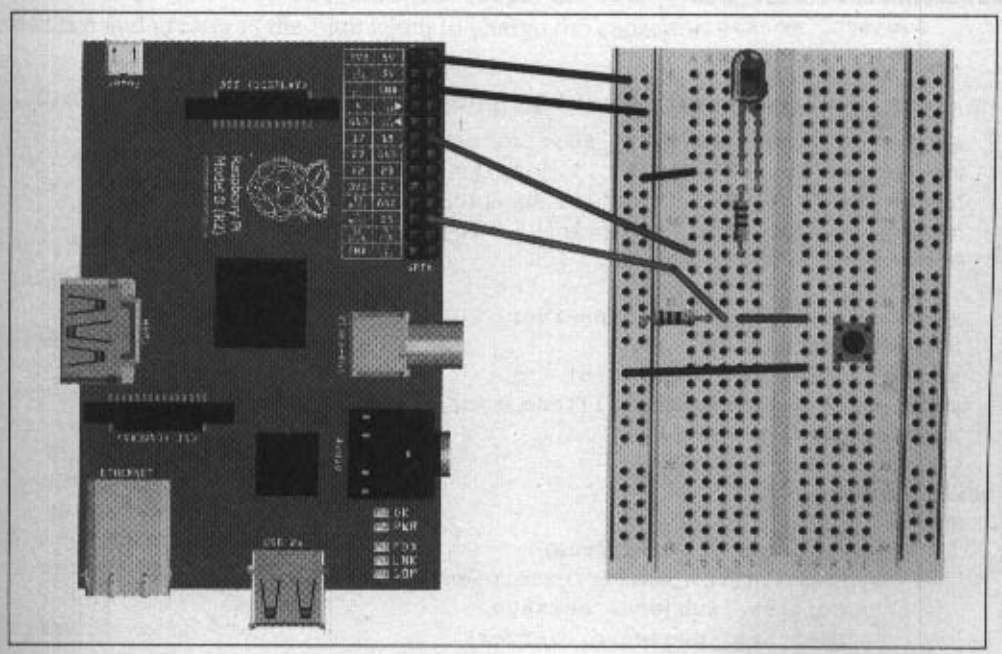


Figure 7.10: Interfacing LED and switch with Raspberry Pi

Box 7.4: Python program for sending an email on switch press

```
import smtplib
from time import sleep
import RPi.GPIO as GPIO
from sys import exit

from_email = '<my-email>'
receipients_list = ['<receipient-email>']
cc_list = []
subject = 'Hello'
message = 'Switch pressed on Raspberry Pi'
username = '<Gmail-username>'
password = '<password>'
server = 'smtp.gmail.com:587'

GPIO.setmode(GPIO.BCM)
GPIO.setup(25, GPIO.IN)


def sendemail(from_addr, to_addr_list, cc_addr_list,
      subject, message,
      login, password,
      smtpserver):

   header = 'From:  %s \n' % from_addr
   header += 'To:  %s \n' % ','.join(to_addr_list)
   header += 'Cc:  %s \n' % ','.join(cc_addr_list)
   header += 'Subject:  %s \n \n' % subject
   message = header + message

   server = smtplib.SMTP(smtpserver)
   server.starttls()
   server.login(login,password)
   problems = server.sendmail(from_addr, to_addr_list, message)
   server.quit()

while True:
   try:
      if (GPIO.input(25) == True):
         sendemail(from_email, receipients_list,
            cc_list, subject, message,
            username, password, server)
      sleep(.01)
```

```
    except KeyboardInterrupt:
        exit()
```

7.6.3 Interfacing a Light Sensor (LDR) with Raspberry Pi

So far you have learned how to interface LED and switch with Raspberry Pi. Now let us look at an example of interfacing a Light Dependent Resistor (LDR) with Raspberry Pi and turning an LED on/off based on the light-level sensed.

Figure 7.11 shows the schematic diagram of connecting an LDR to Raspberry Pi. Connect one side of LDR to 3.3V and other side to a 1μF capacitor and also to a GPIO pin (pin 18 in this example). An LED is connected to pin 18 which is controlled based on the light-level sensed.

Box 7.5 shows the Python program for the LDR example. The *readLDR*() function returns a count which is proportional to the light level. In this function the LDR pin is set to output and low and then to input. At this point the capacitor starts charging through the resistor (and a counter is started) until the input pin reads high (this happens when capacitor voltage becomes greater than 1.4V). The counter is stopped when the input reads high. The final count is proportional to the light level as greater the amount of light, smaller is the LDR resistance and greater is the time taken to charge the capacitor.

■ Box 7.5: Python program for switching LED/Light based on reading LDR reading

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
ldr_threshold = 1000
LDR_PIN = 18
LIGHT_PIN = 25

def readLDR(PIN):
   reading=0
   GPIO.setup(LIGHT_PIN, GPIO.OUT)
   GPIO.output(PIN, False)
   time.sleep(0.1)
   GPIO.setup(PIN, GPIO.IN)
   while (GPIO.input(PIN)==False):
      reading=reading+1
   return reading
```

```
def switchOnLight(PIN):
    GPIO.setup(PIN, GPIO.OUT)
    GPIO.output(PIN, True)

def switchOffLight(PIN):
    GPIO.setup(PIN, GPIO.OUT)
    GPIO.output(PIN, False)


while True:
    ldr_reading = readLDR(LDR_PIN)
    if ldr_reading < ldr_threshold:
        switchOnLight(LIGHT_PIN)
    else:
        switchOffLight(LIGHT_PIN)

    time.sleep(1)
```

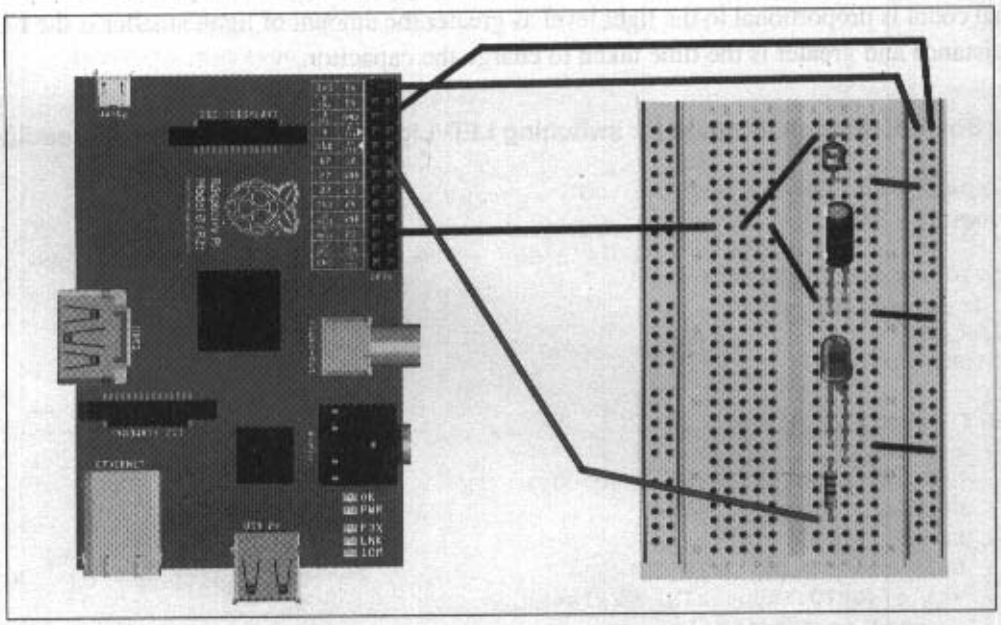


Figure 7.11: Interfacing LDR with Raspberry Pi

7.7 Other IoT Devices

Let us look at single-board mini-computers which are alternatives to Raspberry Pi. Table 7.3 provides a comparison of some single-board mini-computers that can be used for IoT.

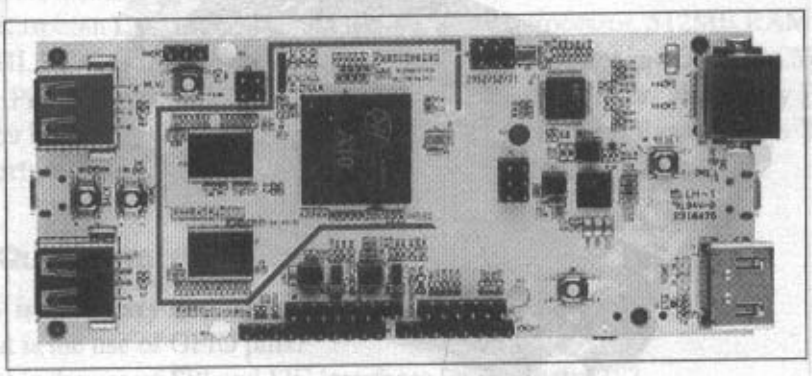


Figure 7.12: pcDuino

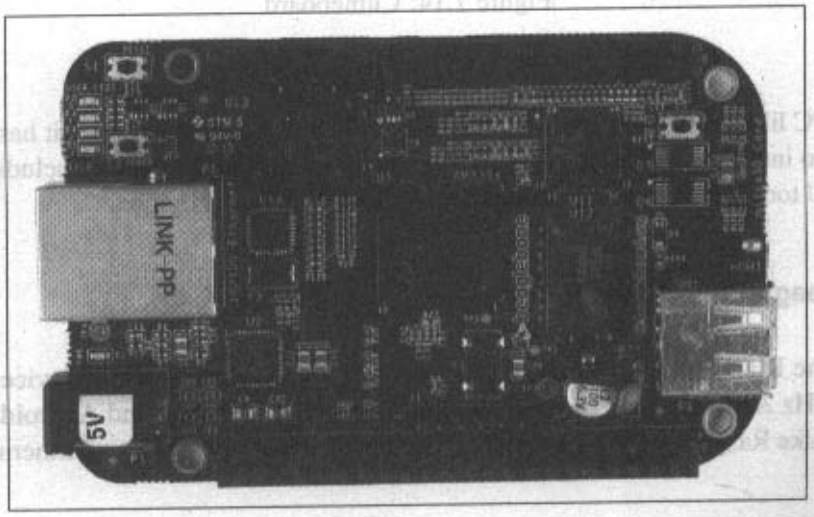


Figure 7.13: Beaglebone Black

7.7.1 pcDuino

pcDuino [105] is an Arduino-pin compatible single board mini-computer that comes with a 1 GHz ARM Cortex-A8 processor. pcDuino is a high performance and cost effective device

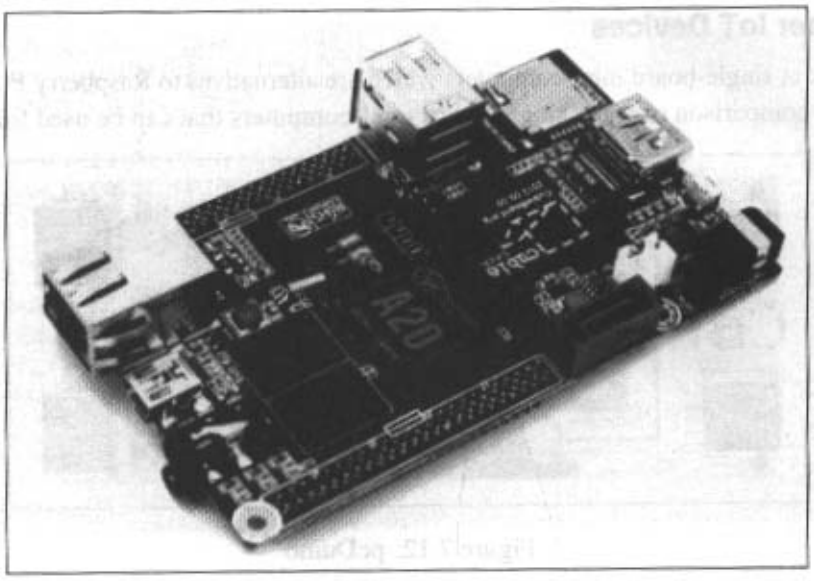


Figure 7.14: Cubieboard

that runs PC like OS such as Ubuntu and Android ICS. Like, Raspberry Pi, it has an HDMI video/audio interface. pcDuino supports various programming languages including C, C++ (with GNU tool chain), Java (with standard Android SDK) and Python.

7.7.2 BeagleBone Black

BeagleBone Black[106] is similar to Raspberry Pi, but a more powerful device. It comes with a 1 GHz ARM Cortex-A8 processor and supports both Linux and Android operating systems. Like Raspberry Pi, it has HDMI video/audio interface, USB and Ethernet ports.

7.7.3 Cubieboard

Cubieboard [107] is powered by a dual core ARM Cortex A7 processor and has a range of input/output interfaces including USB, HDMI, IR, serial, Ethernet, SATA, and a 96 pin extended interface. Cubieboard also provides SATA support. The board can run both Linux and Android operating systems.

Summary

In this chapter you learned about Raspberry Pi which is a low-cost mini-computer. Raspberry Pi supports various flavors of Linux operating system. The official recommended operating system is Raspbian Linux. Raspberry Pi has an ARM processor, 512MB RAM, two USB ports, HDMI, RCA and audio outputs, Ethernet port, SD card slot and DSI and CSI interfaces. Raspberry Pi has serial, SPI and I2C interfaces for data transfer. Raspberry Pi supports Python. You learned how to develop Python programs that run on Raspberry Pi. You learned how to interface LED, switch and LDR with Raspberry Pi.

Review Questions

1. How is Raspberry Pi different from a desktop computer?
2. What is the use of GPIO pins?
3. What is the use of SPI and I2C interfaces on Raspberry Pi?

	Raspberry Pi	pcDuino	BeagleBone Black	Cubieboard
CPU	700 MHz ARM1176JZ-F Processor	1GHz ARM Cortex A8	AM335x 1GHz ARM Cortex-A8	Dual core 1GHz ARM Cortex-A7
GPU	Dual Core VideoCore IV Multimedia Co-Processor	Mali 400	PowerVR SGX530	Dual core ARM Mali 400 MP2
Memory	512MB	1GB	512MB	1GB
Storage	-	2GB Flash (ATmega328)	2GB on-board flash storage	4GB NAND Flash
Networking	10/100M Ethernet	10/100M Ethernet	10/100M Ethernet	10/100M Ethernet
Input/Output	2 USB, SD, MMC, SDIO card slot	-	4+1 USB, MicroSD slot	2 USB, MicroSD slot, SATA, IR sensor
Interfaces	GPIO, SPI, I2C, serial	Serial, ADC, PWM, GPIO, I2C, SPI	69 pin GPIO, SPI, I2C, 4 serial, CAN, GPMC, AIN, MMC, XDMA	96 extend pin interface, including I2C, SPI, RGB/LVDS, CSI/TS, FM-IN, ADC, CVBS, VGA, SPDIF-OUT, R-TP
OS	Rasbian, Pidora, RISC OS, Arch Linux	Ubuntu, Android	Angstrom Linux, Android, Ubuntu	Android, Official Linux distribution
Video	HDMI, Composite RCA (PAL and NTSC)	HDMI	HDMI	HDMI
Audio	3.5mm jack, HDMI	HDMI	HDMI	HDMI
Power	5VDC/700mA	5V/2A	5VDC/460mA	5VDC/2A

Table 7.3: Comparison of single board mini-computers

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.

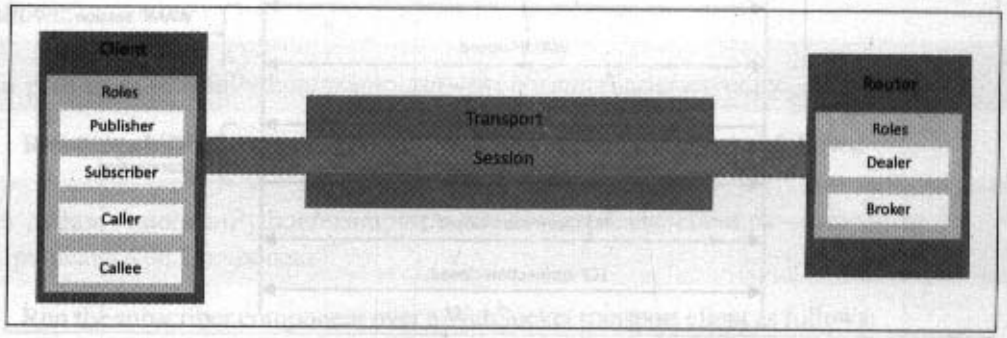


Figure 8.1: WAMP Session between Client and Router

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

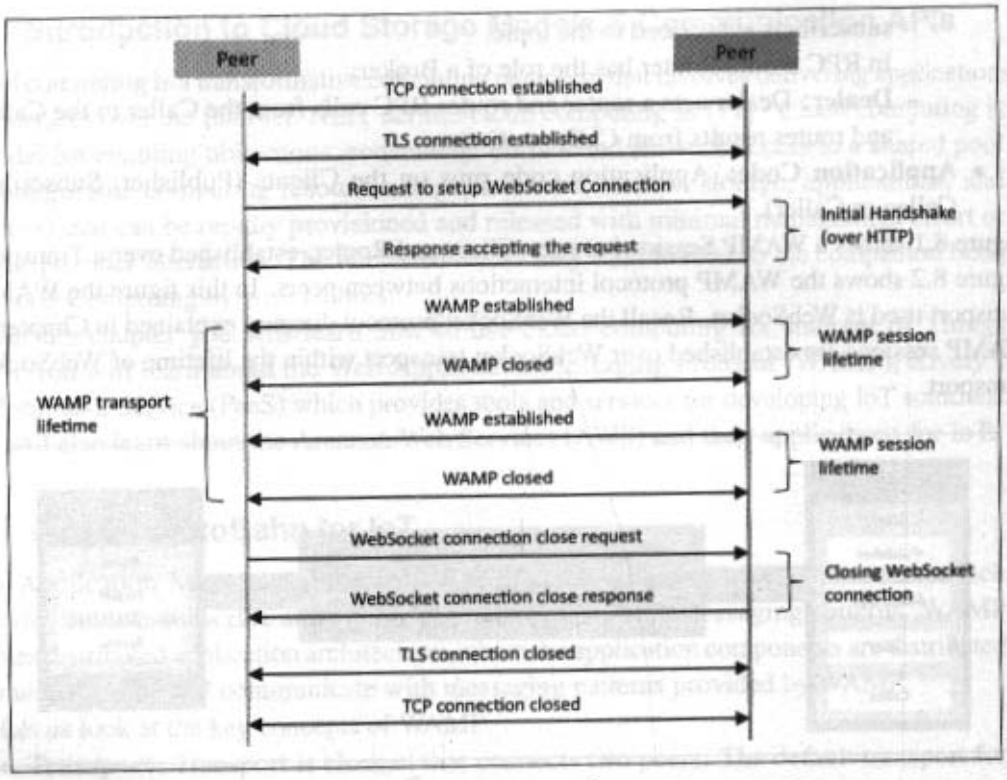


Figure 8.2: WAMP protocol

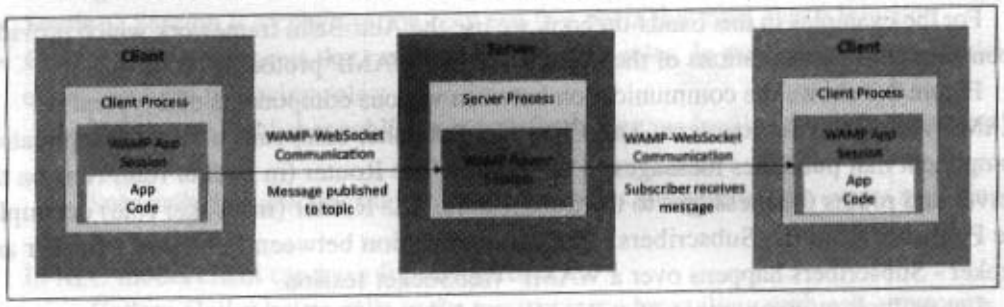


Figure 8.3: Publish-subscribe messaging using WAMP-AutoBahn

```
#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.

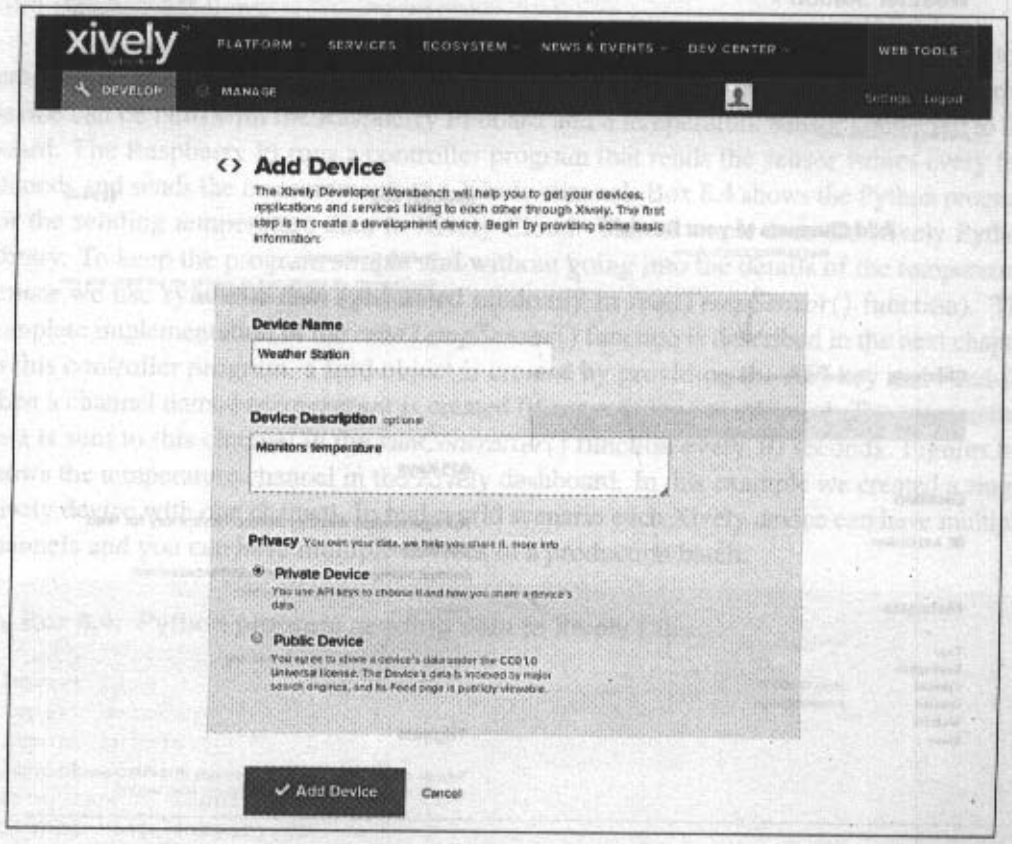


Figure 8.4: Screenshot of Xively dashboard - creating a new device

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

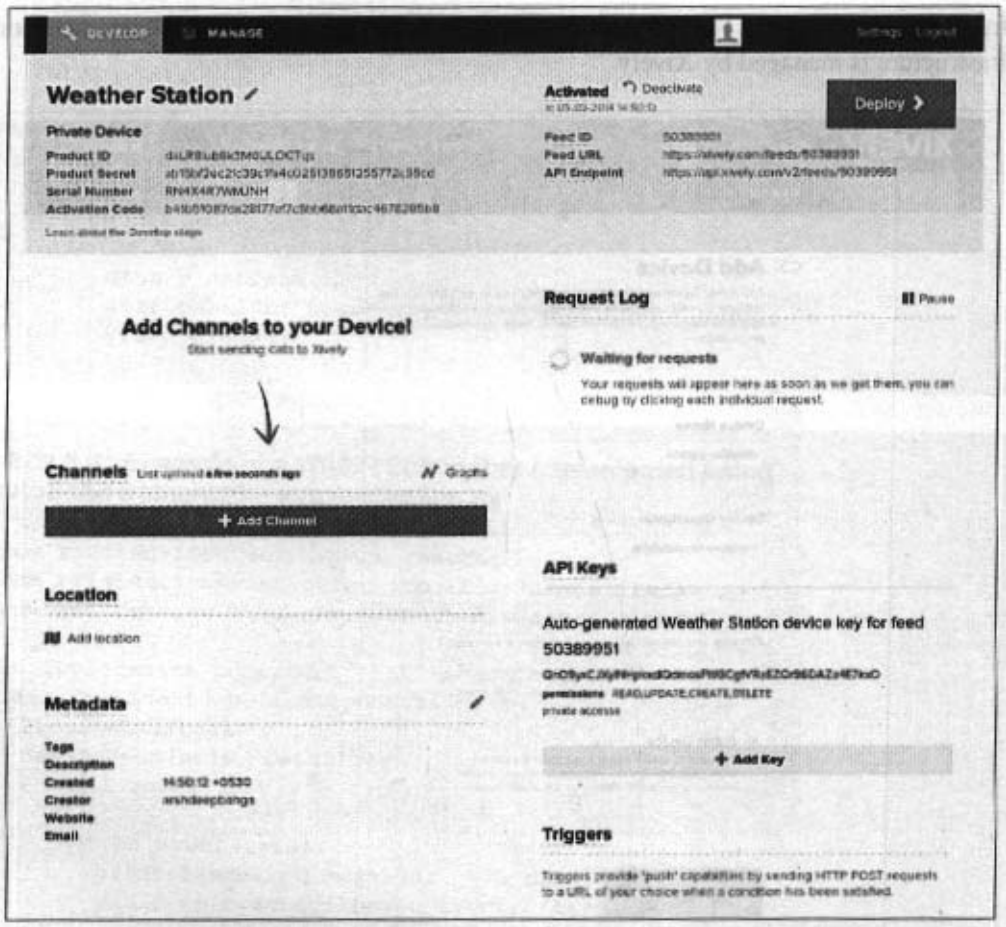


Figure 8.5: Screenshot of Xively dashboard - device details

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

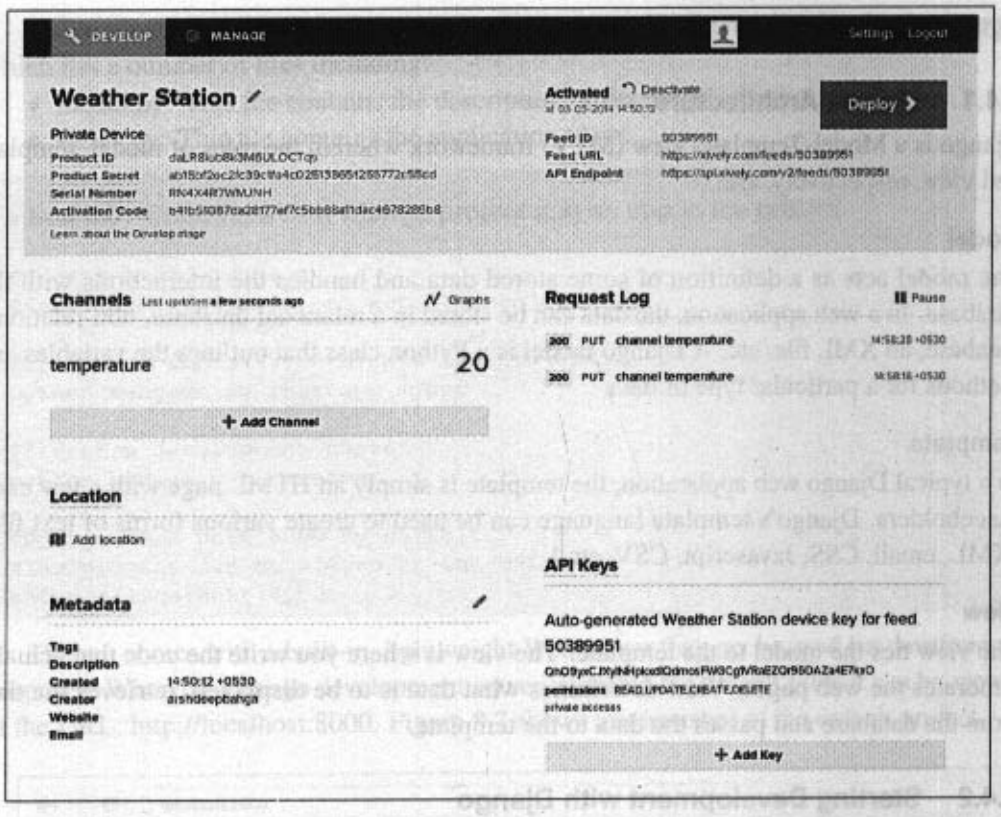


Figure 8.6: Screenshot of Xively dashboard - data sent to channel

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.

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.

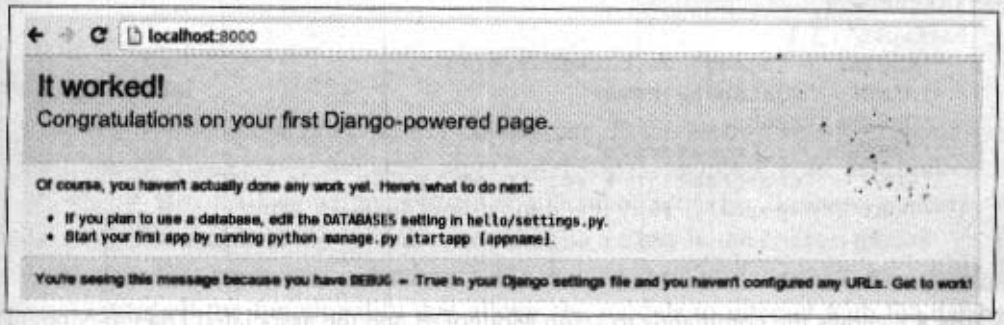


Figure 8.7: Django default project

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].

(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

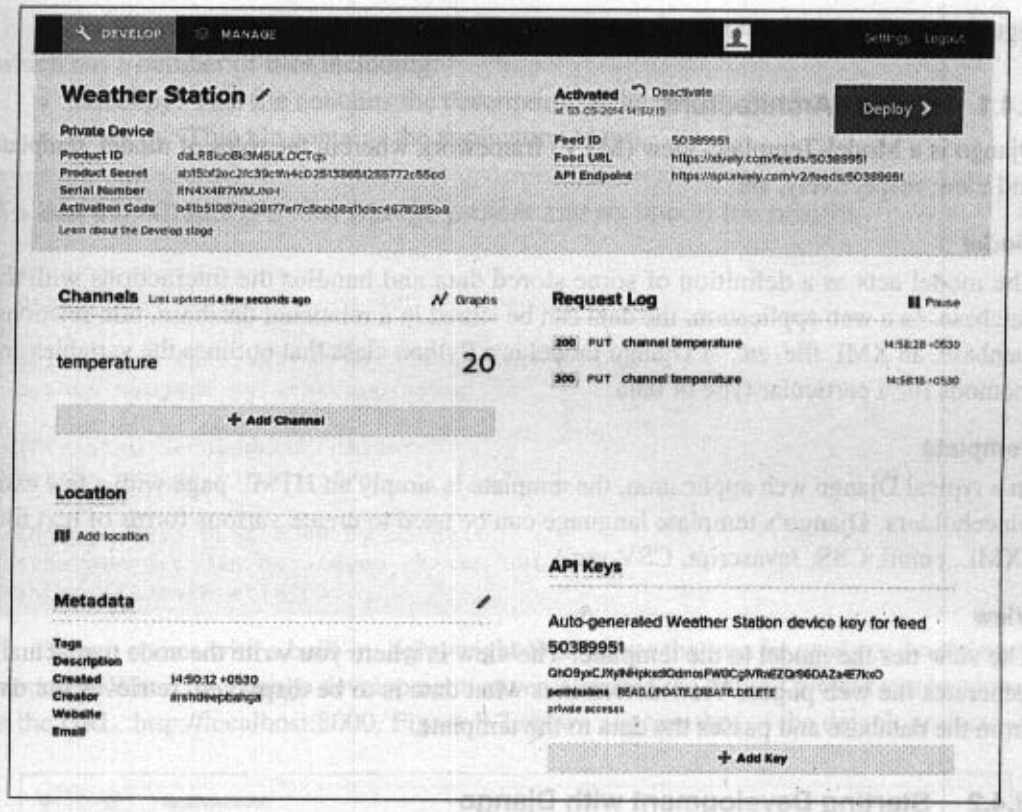


Figure 8.6: Screenshot of Xively dashboard - data sent to channel

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.

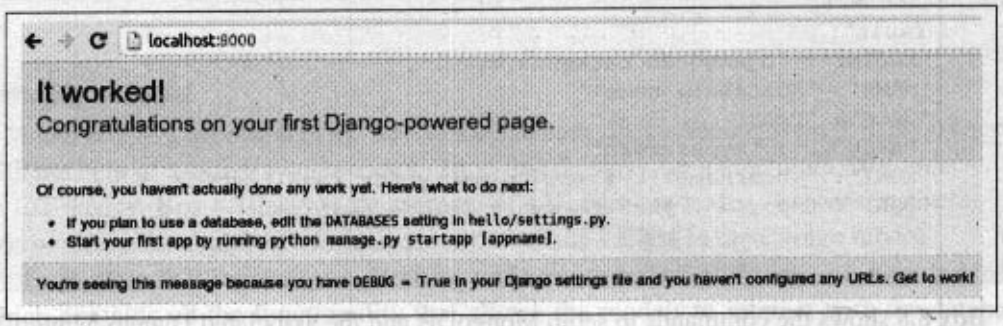


Figure 8.7: Django default project

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

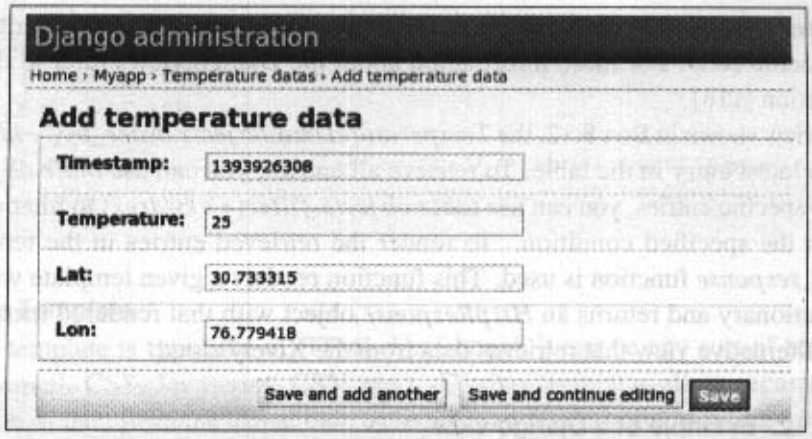


Figure 8.8: Screenshot of default Django admin interface

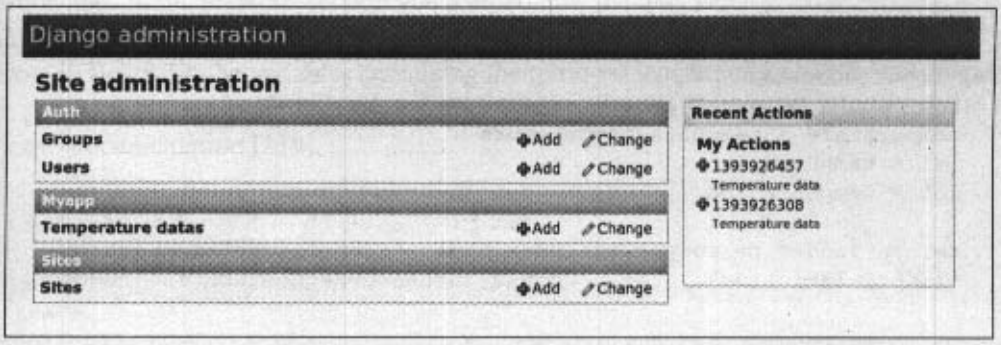


Figure 8.9: Adding data to table from Django admin interface

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

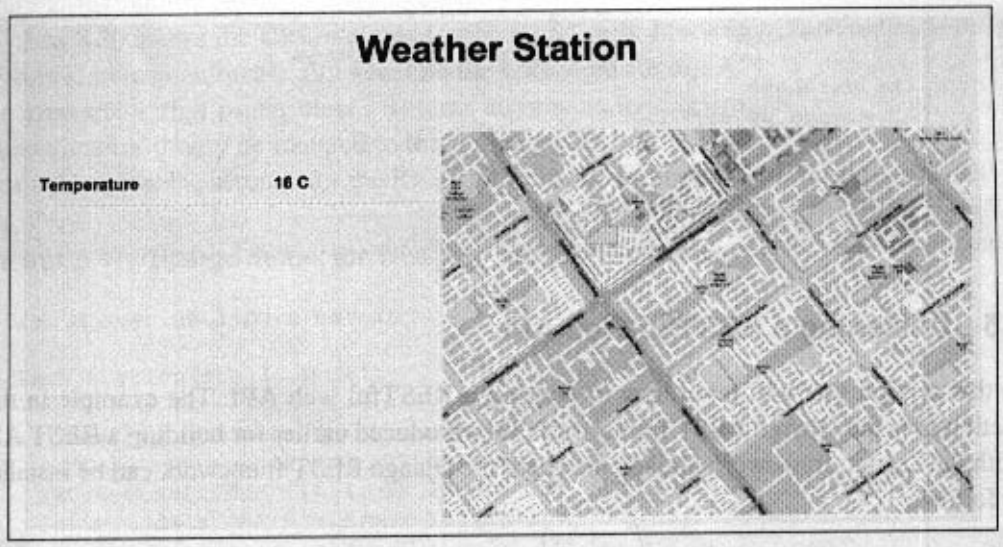


Figure 8.10: Screenshot of a temperature monitoring web application

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.

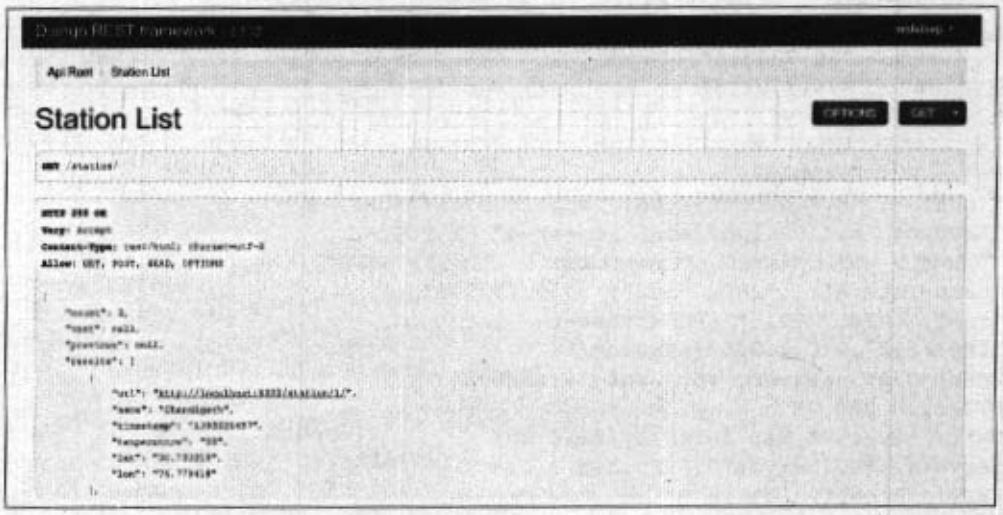


Figure 8.11: Screenshot from the web browsable Station REST API

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.

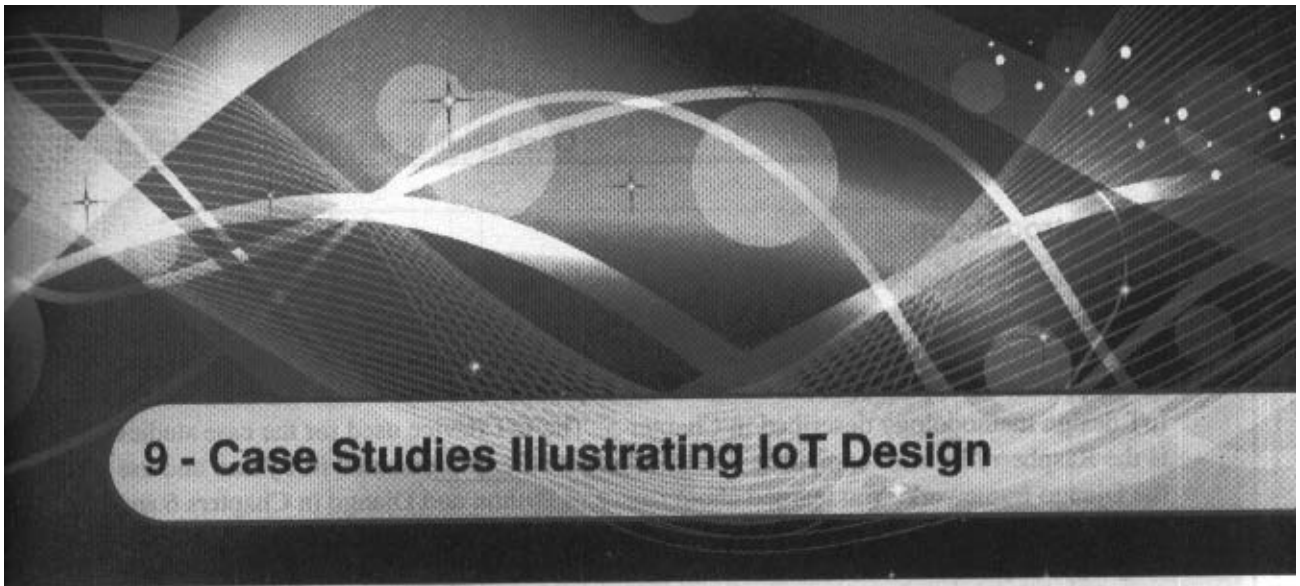
You learned about Django which is an open source web application framework for developing web applications in Python. Django is based on the Model-Template-View architecture. You also learned how to develop a Django application made up of model, view and templates. You learned how to develop a RESTful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



9 - Case Studies Illustrating IoT Design

This Chapter Covers

IoT case studies on:

- Smart Lighting
- Home Intrusion Detection
- Smart Parking
- Weather Monitoring System
- Weather Reporting Bot
- Air Pollution Monitoring
- Forest Fire Detection
- Smart Irrigation
- IoT Printer

9.1 Introduction

In Chapter-2 you learned about the applications of Internet of Things for homes, cities, environment, energy systems, retail, logistics, industry, agriculture and health. This chapter provides concrete implementations of several of these applications helping you understand how sophisticated applications are designed and deployed. The case studies are based on the IoT design methodology described in Chapter-5. The IoT device used for the case studies is the Raspberry Pi mini-computer. The case studies are implemented in Python and use the Django framework. You learned about basics of Python and Django in Chapters 6 and 8, respectively. However, principles you have learned are not limited to these particular languages or platforms.

9.2 Home Automation

9.2.1 Smart Lighting

A design of a smart home automation system was described in Chapter-5 using the IoT design methodology. A concrete
implementation of the system based on Django framework is described in this section. The purpose of the home automation system is to control the lights in a typical home remotely using a web application.

The system includes auto and manual modes. In auto mode, the system measures the light in a room and switches on the light when it gets dark. In manual mode, the system provides the option of manually and remotely switching on/off the light.

Figure 9.1 shows the deployment design of the home automation system. As explained in Chapter-5, the system has two REST services (mode and state) and a controller native service. Figures 9.2 and 9.3 show specifications of the mode and state REST services of the home automation system. The Mode service is a RESTful web service that sets mode to auto or manual (PUT request), or retrieves the current mode (GET request) . The mode is updated to/retrieved from the database. The State service is a RESTful web service that sets the light appliance state to on/off (PUT request), or retrieves the current light state (GET request). The state is updated to/retrieved from the status database.

▪ Box 9.1: Django model for mode and state REST services - models.py

```
from django.db import models

class Mode(models.Model):
   name = models.CharField(max_length=50)
```

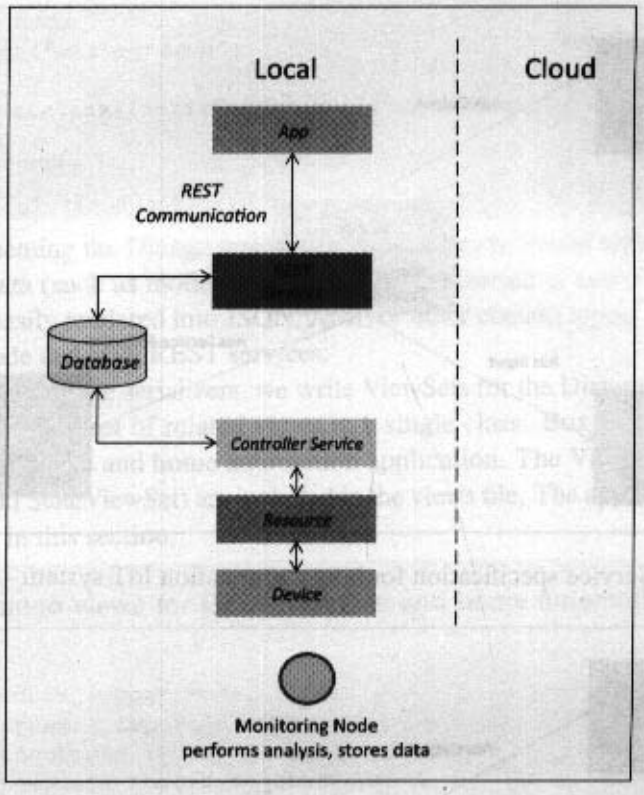


Figure 9.1: Deployment design of the home automation IoT system

```
class State(models.Model):
   name = models.CharField(max_length=50)
```

To start with the implementation of the system, we first map services to Django models. Box 9.1 shows the model fields for the REST services (state - on/off and mode - auto/manual).

Box 9.2: Serializers for mode and state REST services - serializers.py

```
from myapp.models import Mode, State
from rest_framework import serializers

class ModeSerializer(serializers.HyperlinkedModelSerializer):
```

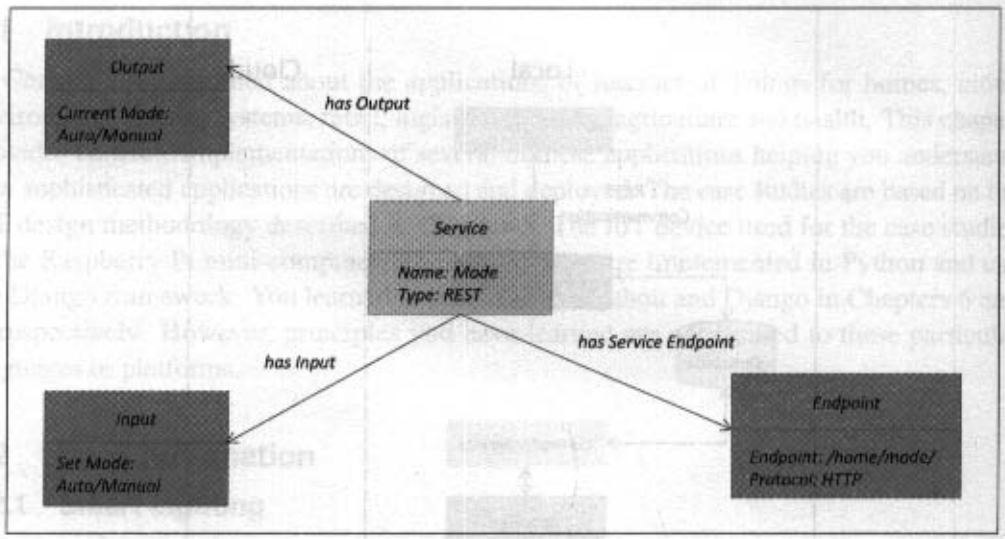


Figure 9.2: Service specification for home automation IoT system - mode service

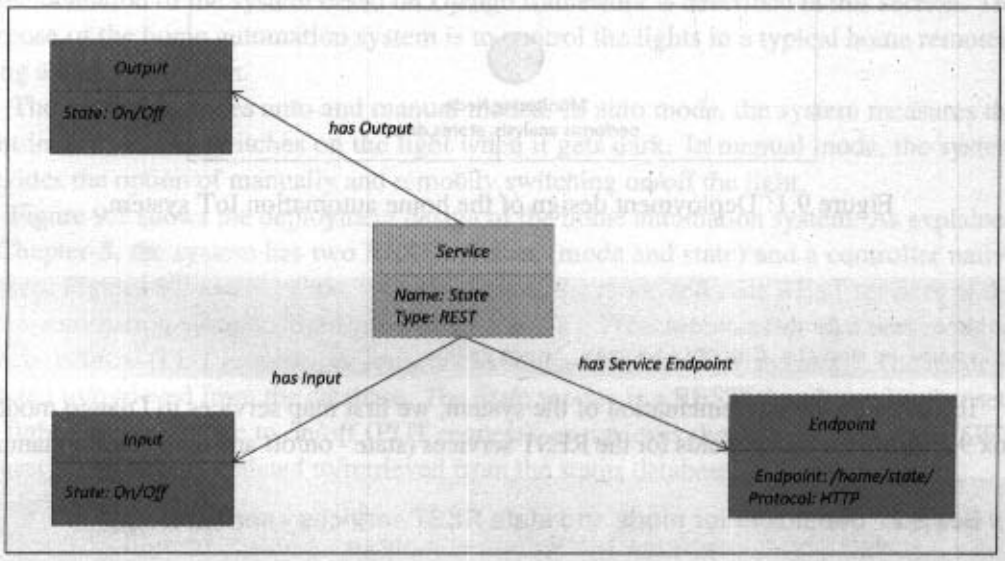


Figure 9.3: Service specification for home automation IoT system - state service

```
class Meta:
```

```
        model = Mode
        fields = ('url', 'name')

class StateSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = State
        fields = ('url', 'name')
```

After implementing the Django model, we implement the model serializers. Serializers allow complex data (such as model instances) to be converted to native Python datatypes that can then be easily rendered into JSON, XML or other content types. Box 9.2 shows the serializers for mode and state REST services.

After implementing the serializers, we write ViewSets for the Django models. ViewSets combine the logic for a set of related views in a single class. Box 9.3 shows the Django views for REST services and home automation application. The ViewSets for the models (ModeViewSet and StateViewSet) are included in the views file. The application view (home) is described later in this section.

▪ Box 9.3: Django views for REST services and home automation application - views.py

```
from myapp.models import Mode, State
from rest_framework import viewsets
from django.shortcuts import render_to_response
from django.template import RequestContext
from myapp.serializers import ModeSerializer, StateSerializer
import requests
import json

class ModeViewSet(viewsets.ModelViewSet):
    queryset = Mode.objects.all()
    serializer_class = ModeSerializer

class StateViewSet(viewsets.ModelViewSet):
    queryset = State.objects.all()
    serializer_class = StateSerializer

def home(request):
    out=''
    currentmode='auto'
    currentstate='off'
```

```
if 'on' in request.POST:
    values = "name":  "on"
    r=requests.put('http://127.0.0.1:8000/state/1/',
    data=values, auth=('myuser', 'password'))
    result=r.text
    output = json.loads(result)
    out=output['name']
if 'off' in request.POST:
    values = "name":  "off"
    r=requests.put('http://127.0.0.1:8000/state/1/',
    data=values, auth=('myuser', 'password'))
    result=r.text
    output = json.loads(result)
    out=output['name']
if 'auto' in request.POST:
    values = "name":  "auto"
    r=requests.put('http://127.0.0.1:8000/mode/1/',
    data=values, auth=('myuser', 'password'))
    result=r.text
    output = json.loads(result)
    out=output['name']
if 'manual' in request.POST:
    values = "name":  "manual"
    r=requests.put('http://127.0.0.1:8000/mode/1/',
    data=values, auth=('myuser', 'password'))
    result=r.text
    output = json.loads(result)
    out=output['name']

r=requests.get('http://127.0.0.1:8000/mode/1/',
auth=('myuser', 'password'))
result=r.text
output = json.loads(result)
currentmode=output['name']

r=requests.get('http://127.0.0.1:8000/state/1/',
auth=('myuser', 'password'))
result=r.text
output = json.loads(result)
currentstate=output['name']

return render_to_response('lights.html','r':out,
'currentmode':currentmode, 'currentstate':currentstate,
context_instance=RequestContext(request))
```

Box 9.4 shows the URL patterns for the REST services and home automation application.

Since ViewSets are used instead of views for the REST services, we can automatically generate the URL configuration by simply registering the viewsets with a router class. Routers automatically determine how the URLs for an application should be mapped to the logic that deals with handling incoming requests.

■ Box 9.4: Django URL patterns for REST services and home automation application - 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'mode', views.ModeViewSet)
router.register(r'state', views.StateViewSet)

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 9.5 shows the code for the Django template for the home automation application.

■ Box 9.5: Django template for home automation application - index.html

```
<!DOCTYPE html>
<html>
<head>

</head>
<body>
<p>{{r}}<p>
<h3>State</h3>
<form action="" method="post">{% csrf_token %}
<input type="submit" name="on" value="on" />
<input type="submit" name="off" value="off" />
</form>
```

```
<br>
<h3>Mode</h3>
<form action="" method="post">{% csrf_token %}
<input type="submit" name="auto" value="auto" />
<input type="submit" name="manual" value="manual" />
</form>

</body>
</html>
```

Figure 9.4 shows a screenshot of the home automation web application.

Figure 9.4: Home automation web application screenshot

Figure 9.5 shows a schematic diagram of the home automation IoT system. The devices and components used in this example are Raspberry Pi mini computer, LDR sensor and relay switch actuator.

Figure 9.6 shows the specification of the controller native service that runs on Raspberry Pi. When in auto mode, the controller service monitors the light level and switches the light on/off and updates the status in the status database. When in manual mode, the controller service, retrieves the current state from the database and switches the light on/off. A Python implementation of the controller service is shown in Box 9.6.

■ Box 9.6: Python code for controller native service - controller.py

```
import time
import datetime
import sqlite3
import spidev
```

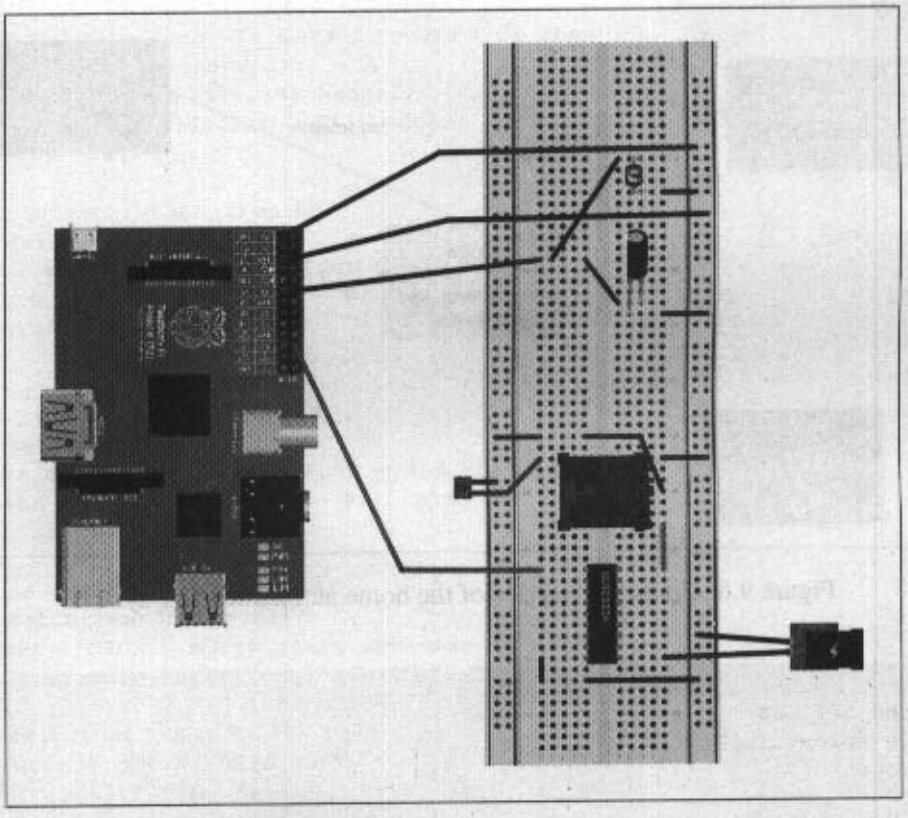


Figure 9.5: Schematic diagram of the home automation IoT system showing the device, sensor and actuator integrated

```
import RPi.GPIO as GPIO

#Initialize SQLite
con = sqlite3.connect('database.sqlite')
cur = con.cursor()

#LDR channel on MCP3008
LIGHT_CHANNEL = 0

#GPIO Setup
GPIO.setmode(GPIO.BCM)
LIGHT_PIN = 25
```

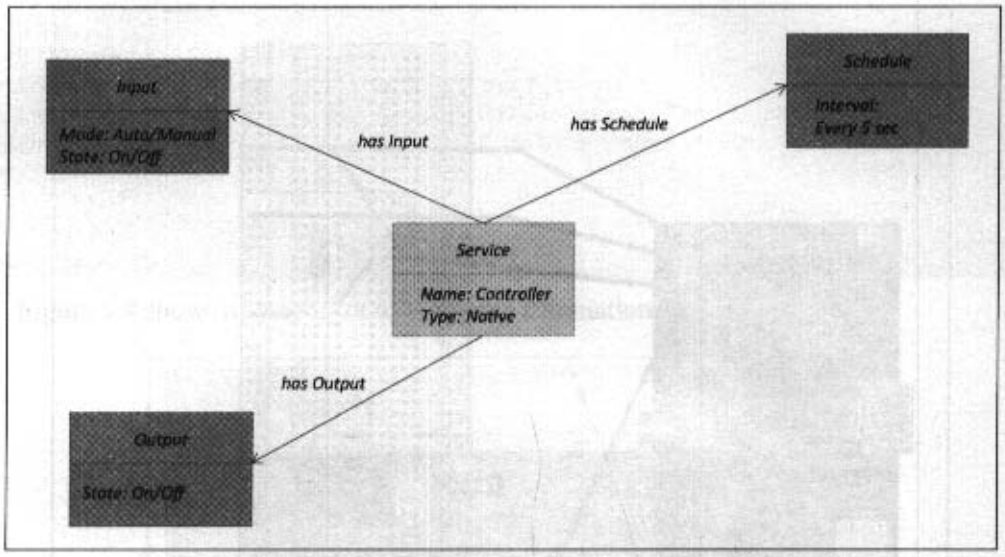


Figure 9.6: Controller service of the home automation IoT system

```
# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)

#Light Level Threshold
threshold=200

#Function to read LDR connected to MCP3008
def readLDR():
   light_level = ReadChannel(LIGHT_CHANNEL)
   lux = ConvertLux(light_level,2)
   return lux

#Function to convert LDR reading to Lux
def ConvertLux(data,places):
   R=10 #10k-ohm resistor connected to LDR
   volts = (data * 3.3) / 1023
   volts = round(volts,places)
   lux=500*(3.3-volts)/(R*volts)
   return lux
```

```
# Function to read SPI data from MCP3008 chip
def ReadChannel(channel):
   adc = spi.xfer2([1,(8+channel)<<4,0])
   data = ((adc[1]&3) << 8) + adc[2]
   return data

#Get current state from DB
def getCurrentMode():
   cur.execute('SELECT * FROM myapp_mode')
   data = cur.fetchone()    #(1, u'auto')
   return data[1]


#Get current state from DB
def getCurrentState():
   cur.execute('SELECT * FROM myapp_state')
   data = cur.fetchone()    #(1, u'on')
   return data[1]

#Store current state in DB
def setCurrentState(val):
   query='UPDATE myapp_state set name="'+val+'"'
   cur.execute(query)

def switchOnLight(PIN):
   GPIO.setup(PIN, GPIO.OUT)
   GPIO.output(PIN, True)

def switchOffLight(PIN):
   GPIO.setup(PIN, GPIO.OUT)
   GPIO.output(PIN, False)

def runManualMode():
   #Get current state from DB
   currentState=getCurrentState()
   if currentState=='on':
      switchOnLight(LIGHT_PIN)
   elif currentState=='off':
      switchOffLight(LIGHT_PIN)

def runAutoMode():
   #Read LDR
   lightlevel=readLDR()

   if lightlevel < ldr_threshold:
```

```
        switchOnLight(LIGHT_PIN)
    else:
        switchOffLight(LIGHT_PIN)

    print 'Manual'+' - '+getCurrentState()

#Controller main function
def runController():
    currentMode=getCurrentMode()
    if currentMode=='auto':
        runAutoMode()
    elif currentMode=='manual':
        runManualMode()

    return true

while True:
    runController()
    time.sleep(5)
```

9.2.2 Home Intrusion Detection

You got an overview of home intrusion detection systems in Chapter-2. A concrete implementation of a home intrusion detection system is described in this section. The purpose of the home intrusion detection system is to detect intrusions using sensors (such as PIR sensors and door sensors) and raise alerts, if necessary.

Figure 9.7 shows the process diagram for the home intrusion detection system. Each room in the home has a PIR motion sensor and each door has a door sensor. These sensors can detect motion or opening of doors. Each sensor is read at regular intervals and the motion detection or door opening events are stored and alerts are sent.

Figure 9.8 shows the domain model for the home intrusion detection system. The domain model includes physical entities for room and door and the corresponding virtual entities. The device in this example is a single-board mini computer which has PIR and door sensors attached to it. The domain model also includes the services involved in the system.

Figure 9.9 shows the information model for the home intrusion detection system. The information model defines the attributes of room and door virtual entities and their possible values. The room virtual entity has an attribute 'motion' and the door virtual entity has an attribute 'state'.

The next step is to define the service specifications for the system. The services are derived from the process specification and the information model. The system has three services - (1) a RESTful web service that retrieves the current state of a door from the

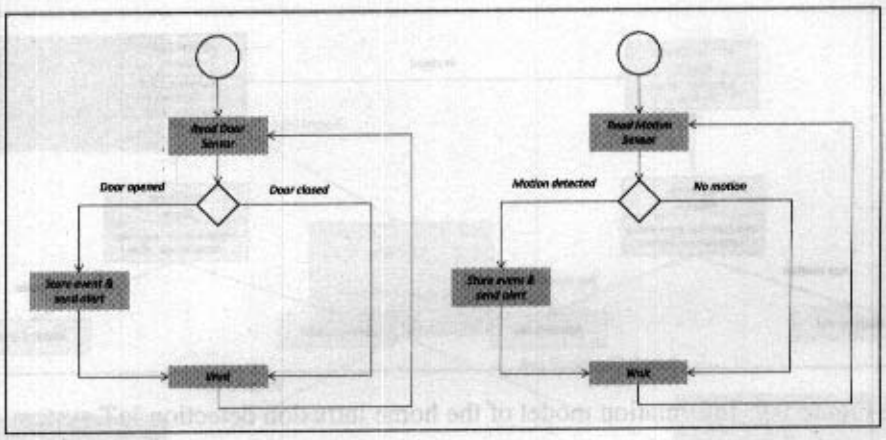


Figure 9.7: Process specification of the home intrusion detection IoT system

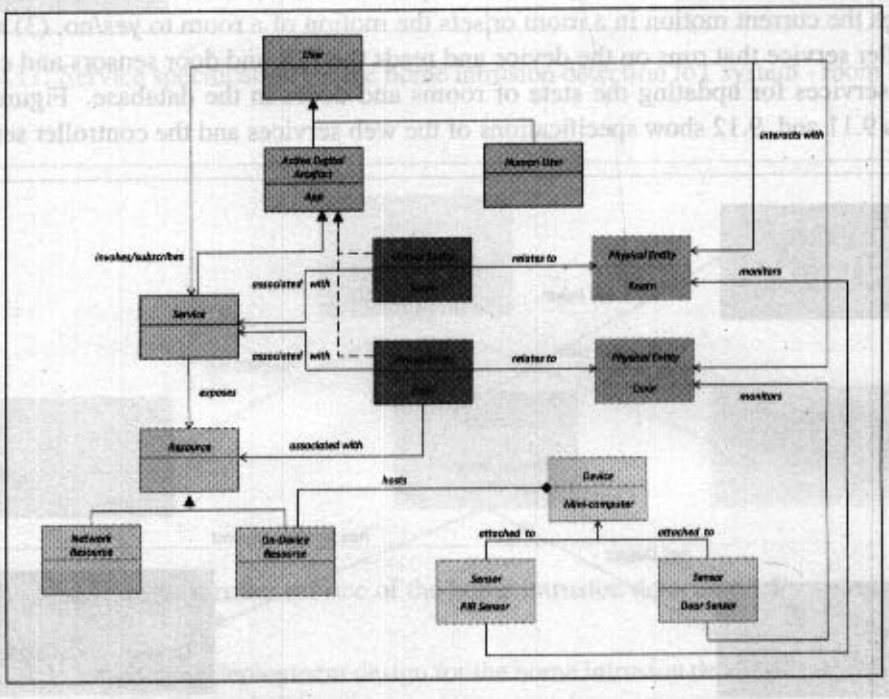


Figure 9.8: Domain model of the home intrusion detection IoT system

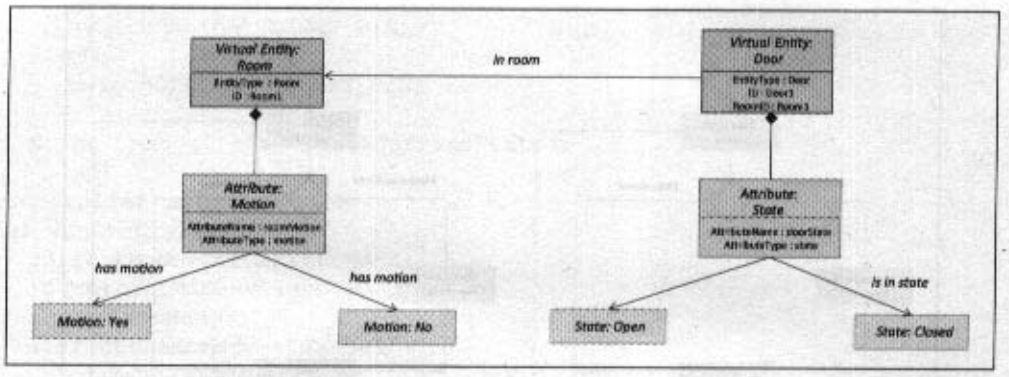


Figure 9.9: Information model of the home intrusion detection IoT system

database or sets the current state of a door to open/closed, (2) A RESTful web service that retrieves the current motion in a room or sets the motion of a room to yes/no, (3) a native controller service that runs on the device and reads the PIR and door sensors and calls the REST services for updating the state of rooms and doors in the database. Figures 9.10, Figures 9.11 and 9.12 show specifications of the web services and the controller service.

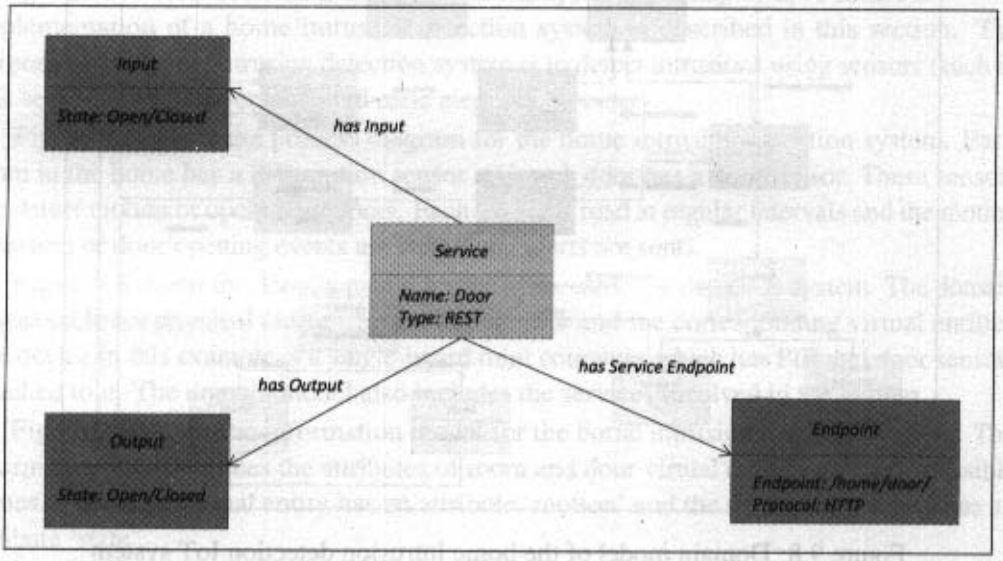


Figure 9.10: Service specification for the home intrusion detection IoT system - door service

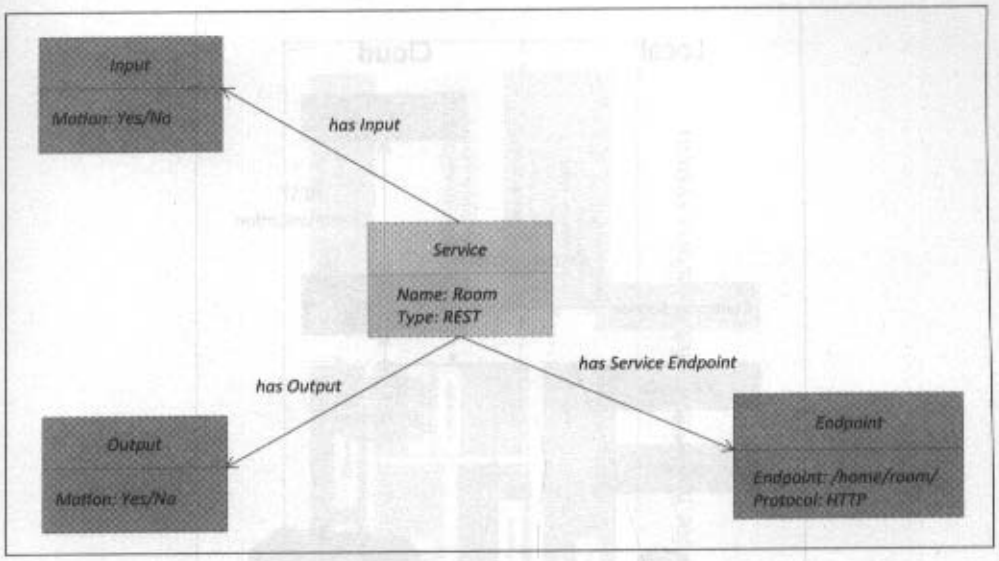


Figure 9.11: Service specification for the home intrusion detection IoT system - room service

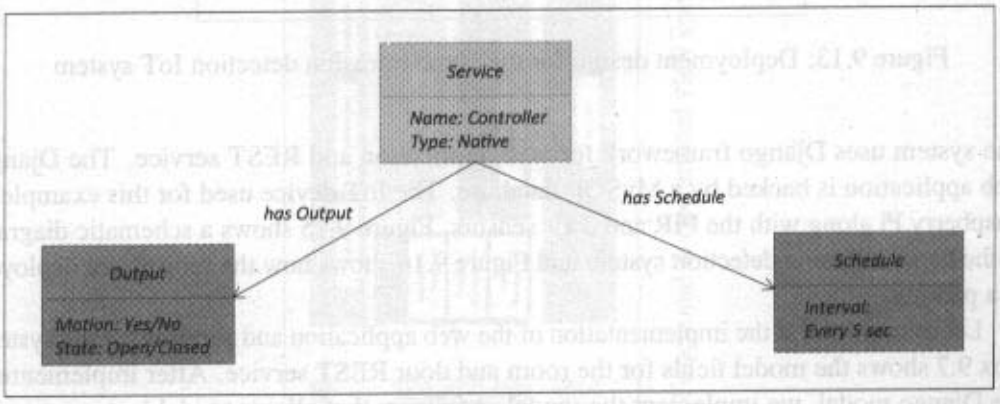


Figure 9.12: Controller service of the home intrusion detection IoT system

Figure 9.13 shows the deployment design for the home intrusion detection system. Recall that this is a level-2 IoT system.

The functional view and the operational view specifications for home intrusion detection system are shown in Figure 9.14. Various options pertaining to the system deployment and operation and their mapping to the corresponding functional groups is shown in Figure 9.14.

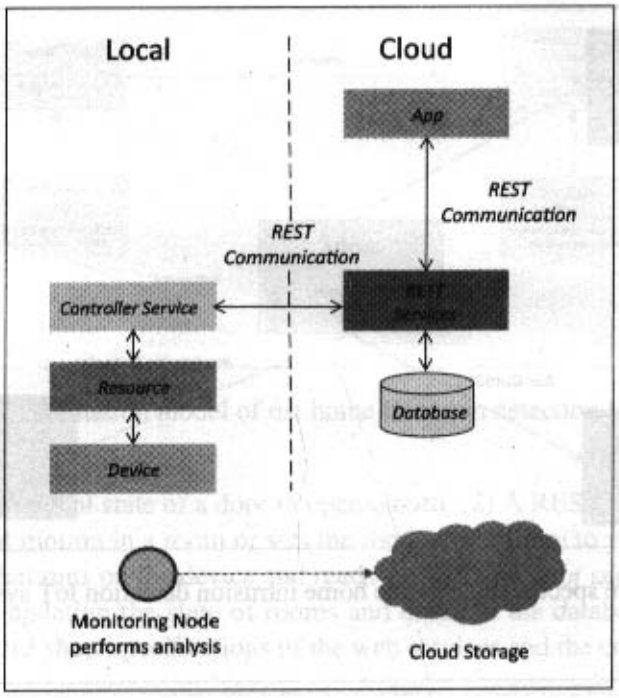


Figure 9.13: Deployment design for the home intrusion detection IoT system

The system uses Django framework for web application and REST service. The Django web application is backed by a MySQL database. The IoT device used for this example is Raspberry Pi along with the PIR and door sensors. Figure 9.15 shows a schematic diagram of the home intrusion detection system and Figure 9.16 shows how the sensors are deployed in a parking.

Let us now look at the implementation of the web application and services for the system. Box 9.7 shows the model fields for the room and door REST service. After implementing the Django model, we implement the model serializers that allows model instances to be converted to native Python datatypes. Box 9.8 shows the serializers for room and door REST services.

Box 9.7: Django model for room and door REST services - models.py

```
from django.db import models
```

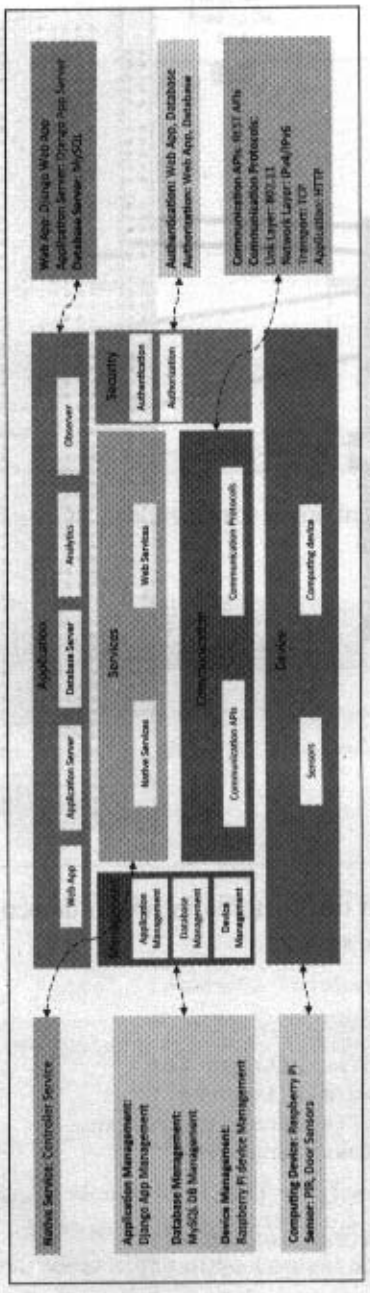


Figure 9.14: Functional & operational view specifications for home intrusion detection system

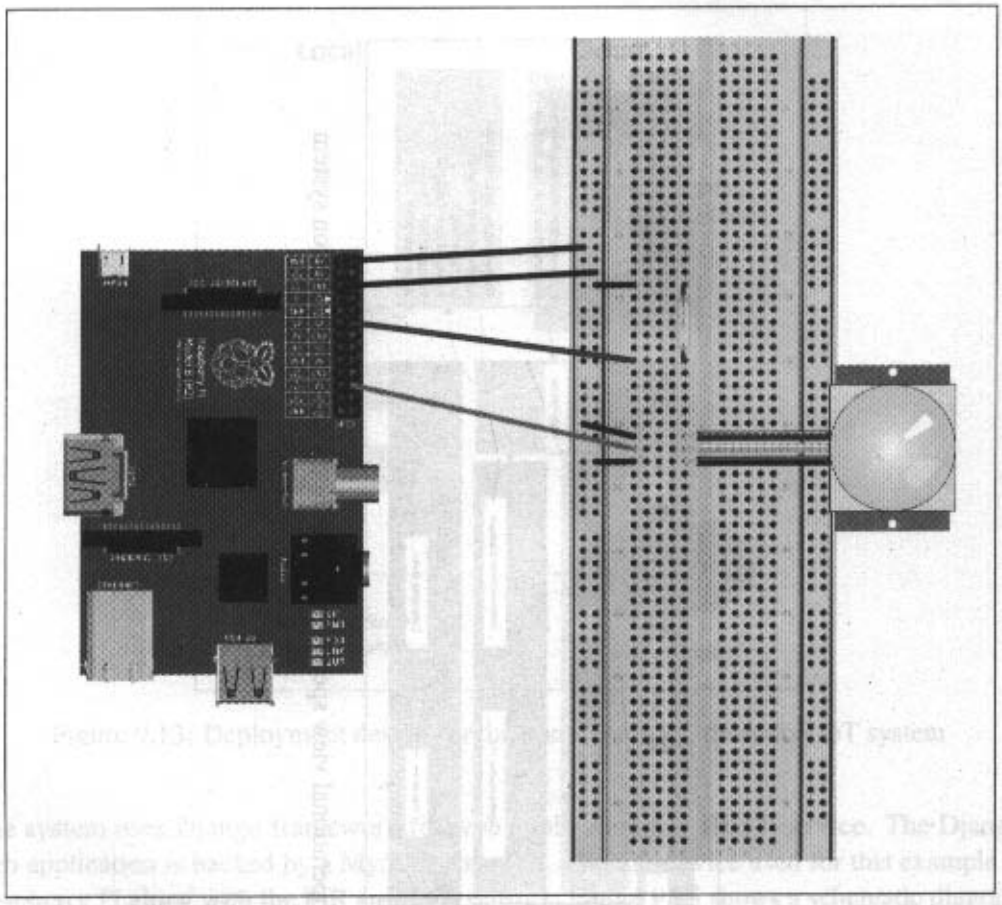


Figure 9.15: Schematic diagram of the home intrusion detection IoT system prototype, showing the device and ultrasonic sensor

```
class Room(models.Model):
    name = models.CharField(max_length=50)
    state = models.CharField(max_length=50)
    timestamp = models.CharField(max_length=50)
    pin = models.CharField(max_length=5)

class Door(models.Model):
    name = models.CharField(max_length=50)
    state = models.CharField(max_length=50)
```

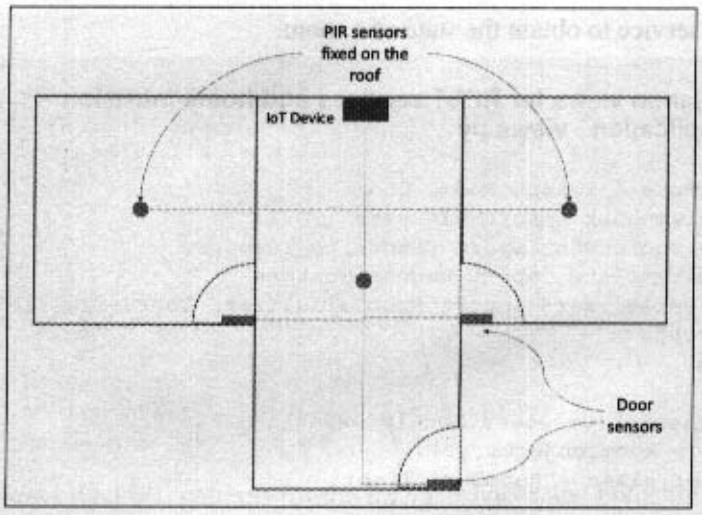


Figure 9.16: Deployment of sensors for home intrusion detection system

```
    timestamp = models.CharField(max_length=50)
    pin = models.CharField(max_length=5)
```

Box 9.8: Serializers for room and door REST services - serializers.py

```
from myapp.models import Room, Door
from rest_framework import serializers

class RoomSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Room
        fields = ('url', 'name', 'state', 'timestamp', 'pin')

class DoorSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Door
        fields = ('url', 'name', 'state', 'timestamp', 'pin')
```

Box 9.9 shows the Django views for REST services and home intrusion detection application. The ViewSets for the models (RoomViewSet and RoomViewSet) are included in the views file. The *home* view renders the content for the home intrusion detection application home page that displays the status of each room. Notice that a request is sent to

the state REST service to obtain the state of a room.

■ Box 9.9: Django views for REST services and home intrusion detection application - views.py

```
from myapp.models import Room, Door
from rest_framework import viewsets
from django.shortcuts import render_to_response
from django.template import RequestContext
from myapp.serializers import RoomSerializer, DoorSerializer
import requests
import json

class RoomViewSet(viewsets.ModelViewSet):
   queryset = Room.objects.all()
   serializer_class = RoomSerializer

class DoorViewSet(viewsets.ModelViewSet):
   queryset = Door.objects.all()
   serializer_class = DoorSerializer

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

   r=requests.get('http://127.0.0.1:8000/door/', auth=('username',
'password'))
   result=r.text
   output = json.loads(result)
   doorCount = output['count']

   roomsDict={}
   for i in range (0, roomCount):
       r=requests.get('http://127.0.0.1:8000/room/'+str(i+1)+'/',
      auth=('username', 'password'))
      result=r.text
      output = json.loads(result)
      roomName=output['name']
      roomState=output['state']
      roomTimestamp=output['timestamp']
      roomsDict[roomName] = [roomState, roomTimestamp]
```

```
    doorsDict={}
    for i in range (0, doorCount):
        r=requests.get('http://127.0.0.1:8000/door/'+str(i+1)+'/',
        auth=('username', 'password'))
        result=r.text
        output = json.loads(result)
        doorName=output['name']
        doorState=output['state']
        doorTimestamp=output['timestamp']
        doorsDict[doorName] = [doorState, doorTimestamp]

    return render_to_response('index.html',
    {'roomsDict':roomsDict, 'doorsDict':doorsDict},
    context_instance=RequestContext(request))
```

Box 9.10 shows the URL patterns for the REST services and home intrusion detection application. Since ViewSets are used instead of views for the REST services, we can automatically generate the URL configuration by simply registering the viewsets with a router class.

▪ Box 9.10: Django URL patterns for REST services and home intrusion detection application - 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'room', views.RoomViewSet)
router.register(r'door', views.DoorViewSet)

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 9.11 shows the code for the Django template for the home intrusion detection

application. This template is rendered by the *home* view. Figure 9.17 shows a screenshot of the home intrusion detection web application.

■ Box 9.11: Django template for home intrusion detection application - index.html

```
<html>
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<title>Home Intrusion Detection App</title>
<link rel="stylesheet" href="/static/css/style.css">
</head>
<body>

<div class="app-container">

<header class="app-header clearfix">
<h1 class="app-logo js-app-title icon-home">Dashboard</h1>
<div class="app-state"><span class="app-loading-loader">
</span></div>
<center>
<h4>Home #:  123</h4>
</center>
</header>

<div role="main" class="app-content clearfix">
<div class="app-loading"><span class="app-loading-loader">
</span></div>
<div class="app-content-inner">

<form class="dashboard-control js-form clearfix">
<fieldset>
<div class="field clearfix">
<center><h4>Rooms</h4></center>
<table width = "90%" border="1">
{% for key,val in roomsDict.items %}
<tr>
<td width="50%">{{key}}
<br>
<center>
{% if val.0 == 'no' %}
<img src="/static/img/g.png">
{% else %}
```

```
<img src="/static/img/r.png">
{% endif %}
<br>
<p>Last Updated:  {{val.1}}</p>
</center>
</td>
</tr>
{% endfor %}
</table>
</div></div>
</fieldset>
</form></div>
<br>

<div role="main" class="app-content clearfix">
<div class="app-loading"><span class="app-loading-loader">
</span></div>
<div class="app-content-inner">

<form class="dashboard-control js-form clearfix">
<fieldset>
<div class="field clearfix">
<center><h4>Doors</h4></center>
<table width = "90%" border="1">
{% for key,val in doorsDict.items %}
<tr>
<td width="50%">{{key}}
<br>
<center>
{% if val.0 == 'closed' %}
<img src="/static/img/g.png">
{% else %}
<img src="/static/img/r.png">
{% endif %}
<br>
<p>Last Updated:  {{val.1}}</p>
</center>
</td>
</tr>
{% endfor %}
</table>
</div>
</div>
</fieldset>
</form>
```

```
</div>
</div>
<script src="https://ajax.googleapis.com/ajax/libs/
jquery/1.8.2/jquery.min.js"></script>
<script src="/static/js/script.js"></script>

</body> </html>
```

A Python implementation of the controller service native service that runs on Raspberry Pi, is shown in Box 9.12. The *runController* function is called every second and the readings of the PIR and door sensors are obtained. The current states of the room and door are then updated by sending a PUT request to the corresponding REST services.

■ Box 9.12: Python code for controller native service - intrusion detection - controller.py

```
import RPi.GPIO as GPIO
import time
import sys

GPIO.setmode(GPIO.BCM)

global PIR_SENSOR_PIN

global DOOR_SENSOR_PIN
def readingPIRSensor():

   if GPIO.input(PIR_SENSOR_PIN):
      return 1
   else:
      return 0

def readingDoorSensor():
   if GPIO.input(DOOR_SENSOR_PIN):
      return 1
   else:
      return 0

def runController():
   pirState = readingPIRSensor()
   if pinState == 1:
      setPIRState('yes')
   else:
```

```
        setPIRState('no')

    doorState = readingDoorSensor()
    if doorState == 1:
        setDoorState('open')
    else:
        setDoorState('closed')

def setPIRState(val):
    values = {"state":  val, "timestamp":  str(time.time())}
    r=requests.put('http://127.0.0.1:8000/room/1/', data=values,
    auth=('username', 'password'))
def setDoorState(val):
    values = {"state":  val, "timestamp":  str(time.time())}
    r=requests.put('http://127.0.0.1:8000/door/1/', data=values,
    auth=('username', 'password'))

def setupController():
    r=requests.get('http://127.0.0.1:8000/config/1/', data=values,
    auth=('username', 'password'))
    configStr=r.text
    config = json.loads(configStr)
    global PIR_SENSOR_PIN = config['room1']
    global DOOR_SENSOR_PIN = config['door1']
    GPIO.setup(PIR_SENSOR_PIN,GPIO.IN)
    GPIO.setup(DOOR_SENSOR_PIN,GPIO.IN, pull_up_down=GPIO.PUD_UP)

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

9.3 Cities

9.3.1 Smart Parking

You got an overview of smart parking systems in Chapter-2. A concrete implementation of a smart parking IoT system is described in this section.

The purpose of a smart parking system is to detect the number of empty parking slots and send the information over the Internet to smart parking application backends. These applications can be accessed by drivers from smartphones, tablets or from in-car navigation systems. In smart parking, sensors are used for each parking slot, to detect whether the slot is empty or occupied. This information is aggregated by a local controller and then sent over

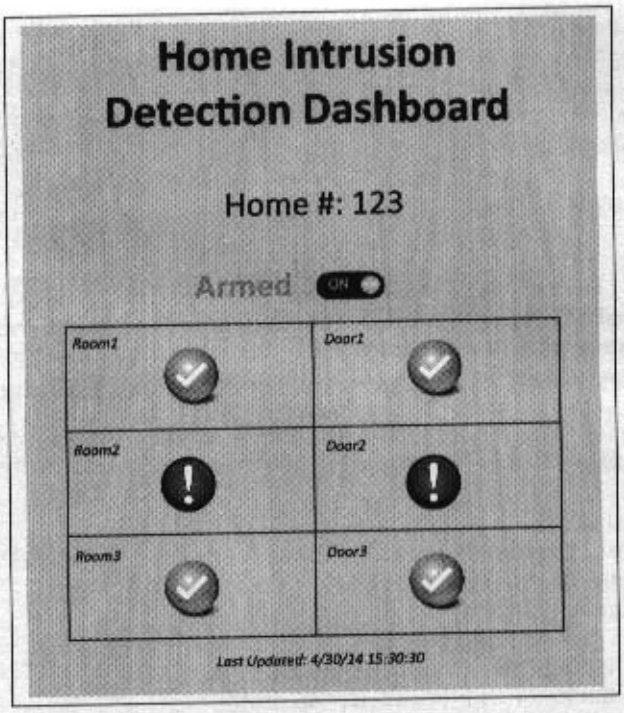


Figure 9.17: Home intrusion detection web application screenshot

the Internet to a server.

Figure 9.18 shows the process diagram for the smart parking system. Each parking slot has an ultrasonic sensor fixed above, which can detect the presence of a vehicle in the slot. Each sensor is read at regular intervals and the state of the parking slot (empty or occupied) is updated in a database.

Figure 9.19 shows the domain model for the smart parking system. The domain model includes a physical entity for the parking slot and the corresponding virtual entity. The device in this example is a single-board mini computer which has ultrasonic sensor attached to it. The domain model also includes the services involved in the system.

Figure 9.20 shows the information model for the smart parking system. The information model defines the attribute (state) of the parking slot virtual entity with two possible values (empty or occupied).

The next step is to define the service specifications for the system. The services are derived from the process specification and the information model. The smart parking system has two services - (1) a service that monitors the parking slots (using ultrasonic sensors) and

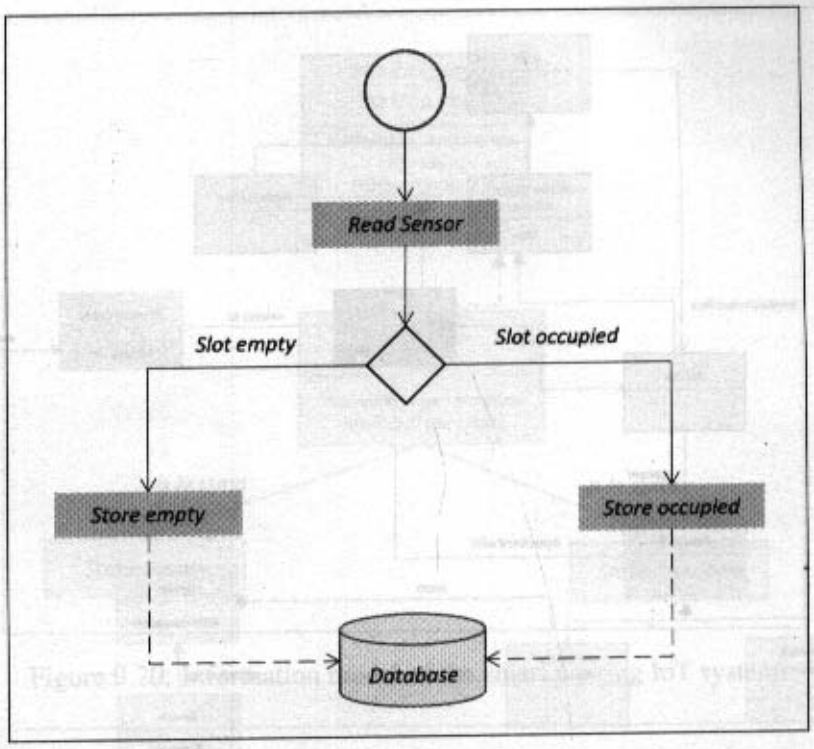


Figure 9.18: Process specification for the smart parking IoT system

updates the status in a database on the cloud (REST web service), (2) a service that retrieves the current state of the parking slots (controller native service). Figures 9.21 and 9.22 show specifications of the controller and state services of the smart parking system.

The functional view and the operational view specifications for smart parking system are similar to the specifications for the home intrusion detection system shown in Figure 9.14. The system uses Django framework for web application and REST service - both of which you learned about from earlier chapters of this book. The Django web application is supported by a MySQL database. The IoT device used for this example is Raspberry Pi along with the ultrasonic sensors. Figure 9.23 shows how the sensors are deployed in a parking and Figure 9.24 shows a schematic diagram of the smart parking system.

Let us now look at the implementation of the web application and services for the smart parking system. Box 9.13 shows the model fields for the state REST service. After implementing the Django model, we implement the model serializer which allows model

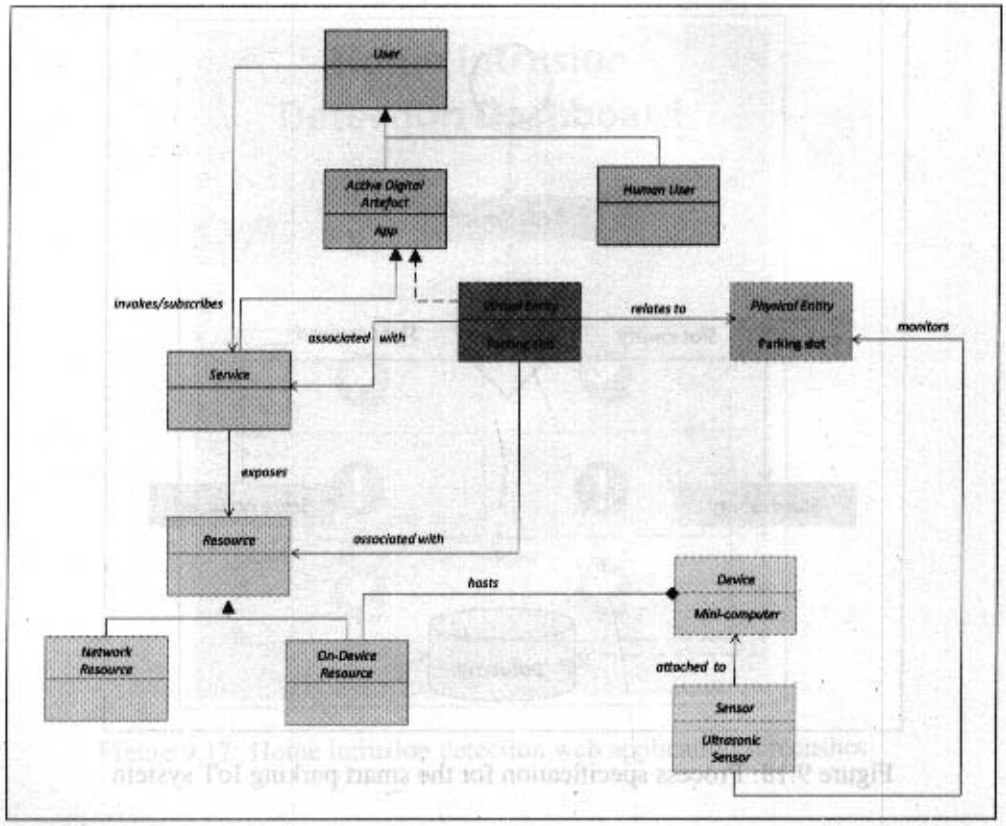


Figure 9.19: Domain model of the smart parking IoT system

instances to be converted to native Python datatypes. Box 9.14 shows the serializer for state REST service.

Box 9.13: Django model for REST service - models.py

```
from django.db import models

class State(models.Model):
   name = models.CharField(max_length=50)
```

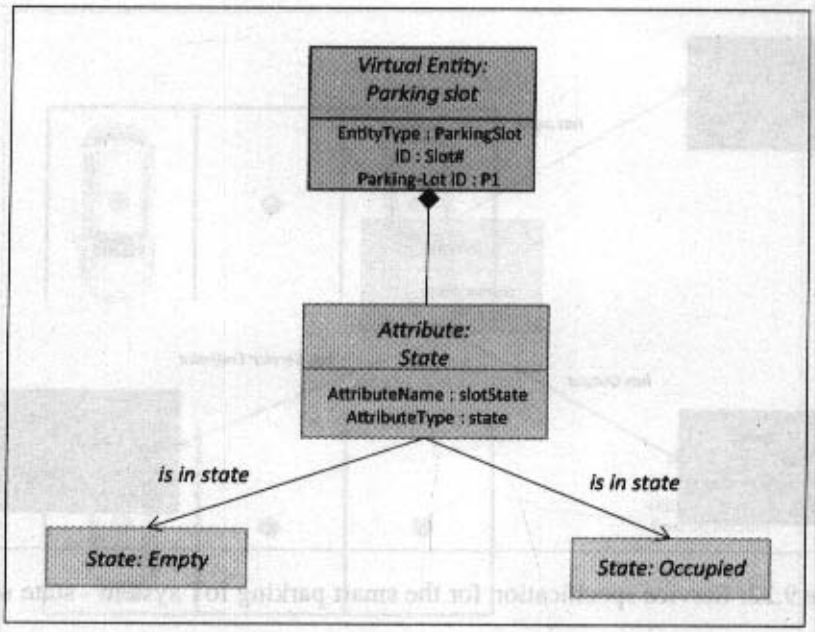


Figure 9.20: Information model of the smart parking IoT system

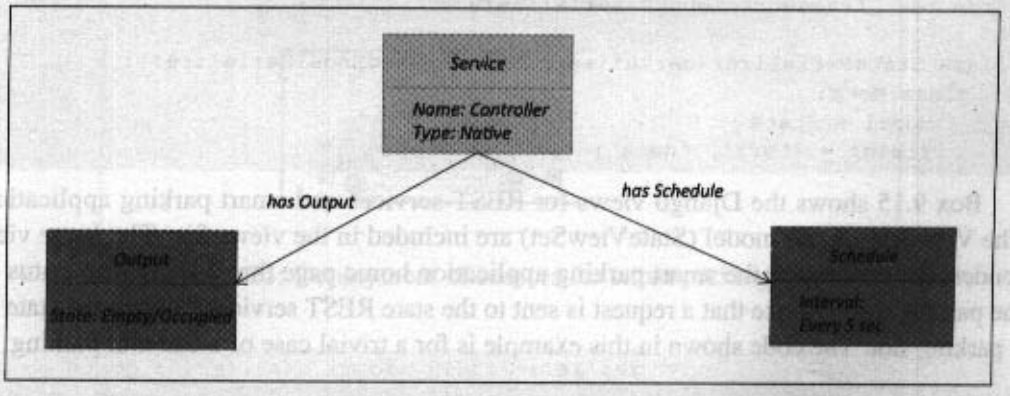


Figure 9.21: Controller service of the smart parking IoT system

▪ Box 9.14: Serializer for REST service - serializers.py

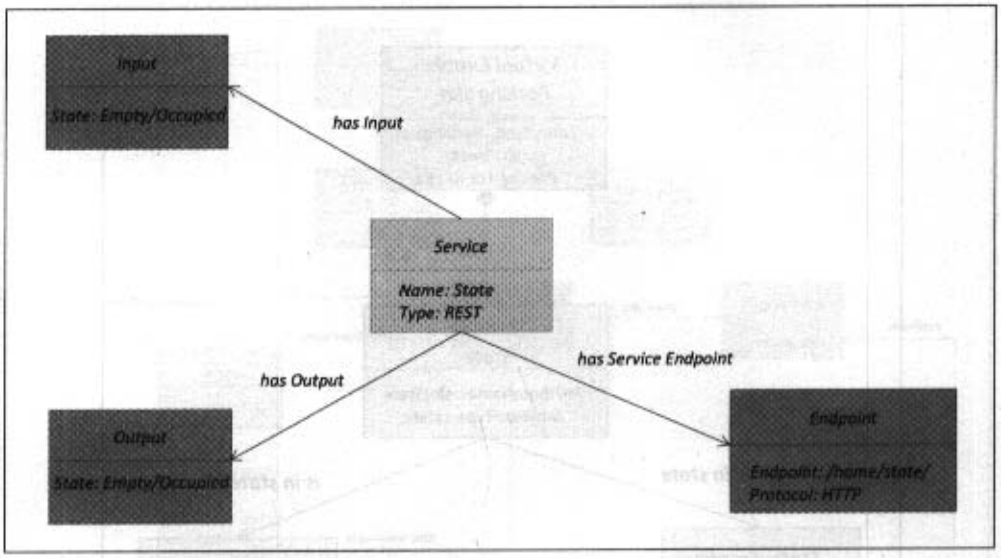


Figure 9.22: Service specification for the smart parking IoT system - state service

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

class StateSerializer(serializers.HyperlinkedModelSerializer):
   class Meta:
      model = State
      fields = ('url', 'name')
```

Box 9.15 shows the Django views for REST services and smart parking application. The ViewSets for the model (StateViewSet) are included in the views file. The *home* view renders the content for the smart parking application home page that displays the status of the parking slots. Notice that a request is sent to the state REST service to obtain the state of a parking slot. The code shown in this example is for a trivial case of a one-slot parking.

■ Box 9.15: Django views for REST service and smart parking application - views.py

```
from myapp.models import State
from rest_framework import viewsets
from django.shortcuts import render_to_response
from django.template import RequestContext
```

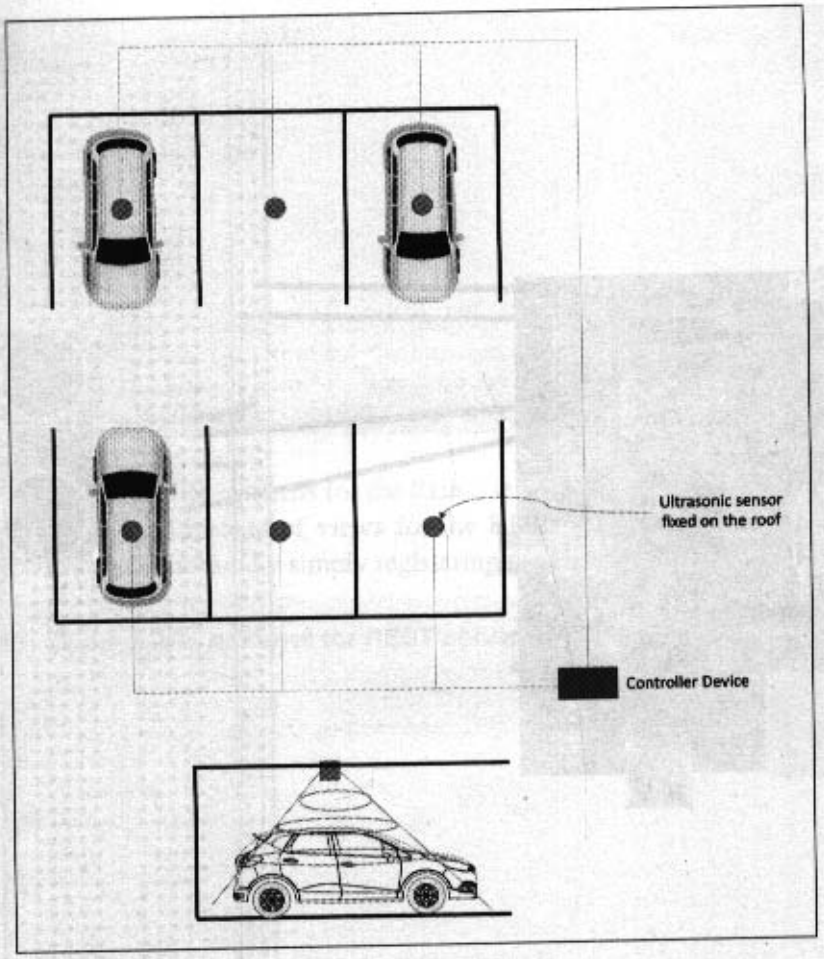


Figure 9.23: Deployment of sensors for smart parking system

```
from myapp.serializers import StateSerializer
import requests
import json

class StateViewSet(viewsets.ModelViewSet):
    queryset = State.objects.all()
    serializer_class = StateSerializer

def home(request):
```

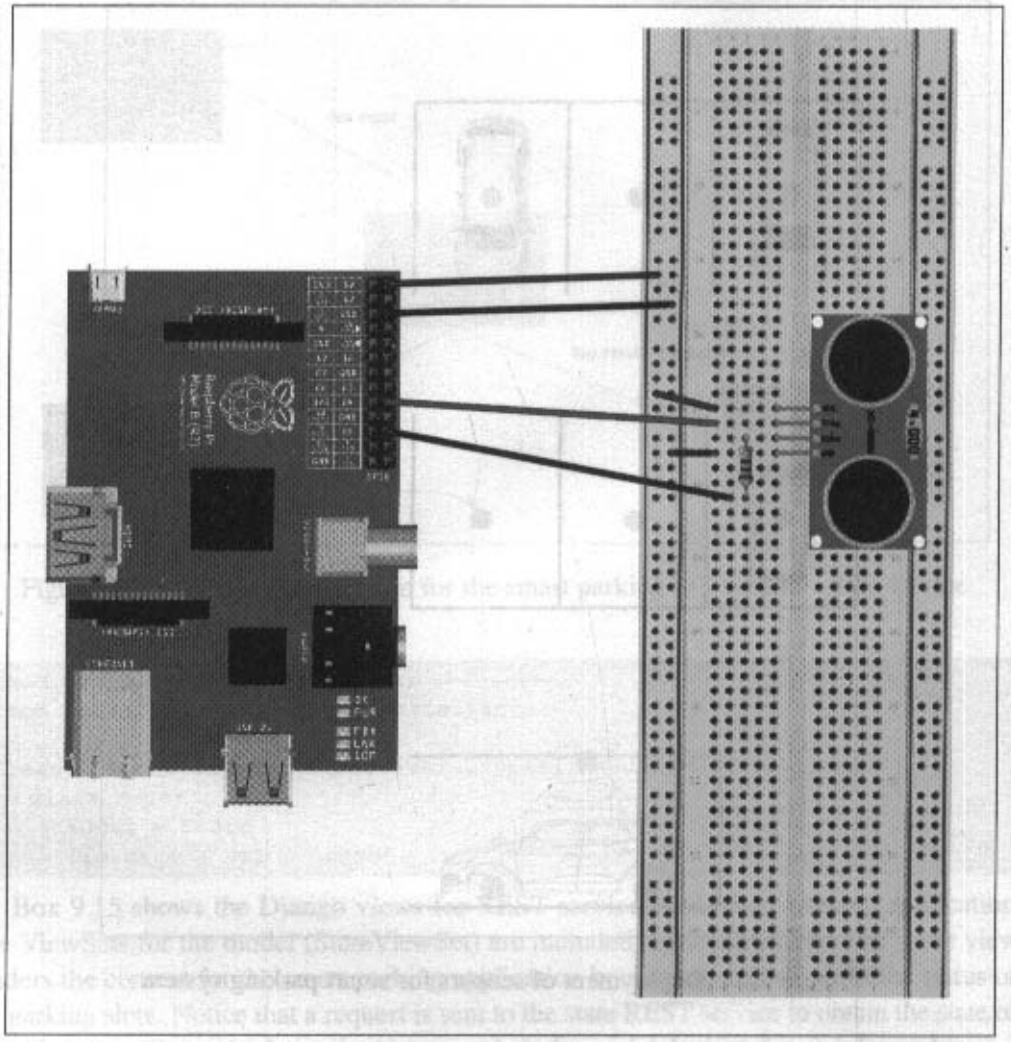


Figure 9.24: Schematic diagram of the smart parking IoT system prototype, showing the device and ultrasonic sensor

```
    currentstate='off'
r=requests.get('http://127.0.0.1:8000/state/1/',
auth=('username', 'password'))
result=r.text
```

```
    output = json.loads(result)
    currentstate=output['name']

    if currentstate=='empty':
       occupiedCount=0
       emptyCount=1
    else:
       occupiedCount=1
       emptyCount=0

    return render_to_response('index.html',
    {'currentstate':currentstate, 'occupiedCount':
    occupiedCount, 'emptyCount':  emptyCount},
    context_instance=RequestContext(request))
```

Box 9.16 shows the URL patterns for the REST service and smart parking application. Since ViewSets are used instead of views for the REST service, we can automatically generate the URL configuration by simply registering the viewsets with a router class.

■ Box 9.16: Django URL patterns for REST service and smart parking application - 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'state', views.StateViewSet)

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 9.17 shows the code for the Django template for the smart parking application. This template is rendered by the *home* view.

Box 9.17: Django template for smart parking application - index.html

```
<html>
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<title>Smart Parking App</title>
<link rel="stylesheet" href="/static/css/style.css">
</head>
<body>
<div class="app-container">
<header class="app-header clearfix">
<h1 class="app-logo js-app-title icon-home">Smart Parking
Dashboard</h1>
<div class="app-state"><span
class="app-loading-loader"></span></div>
<center>
<h4>Parking Lot #:  123</h4>
<h4>Empty:  {{emptyCount}} Occupied:  {{occupiedCount}}</h4>
</center>
</header>
<div role="main" class="app-content clearfix">
<div class="app-loading"><span class="app-loading-loader"></span></div>
<div class="app-content-inner">
<form class="dashboard-control js-form clearfix">
<fieldset>

<div class="field clearfix">
<table width = "90%" border="1">
<tr>
<td width="100%">1
<br>
<center>
{% if currentstate == 'empty' %}
<img src="/static/img/empty.png">
{% else %}
<img src="/static/img/occupied.png">
{% endif %}
</center>
</td>
</tr>
</table>
</div>
</div>
```

```
</fieldset>
</form>
</div></div></div>

<script src="https://ajax.googleapis.com/ajax/libs/
jquery/1.8.2/jquery.min.js"></script>
<script src="/static/js/script.js"></script>
</body></html>
```

Figure 9.25 shows a screenshot of the smart parking web application.

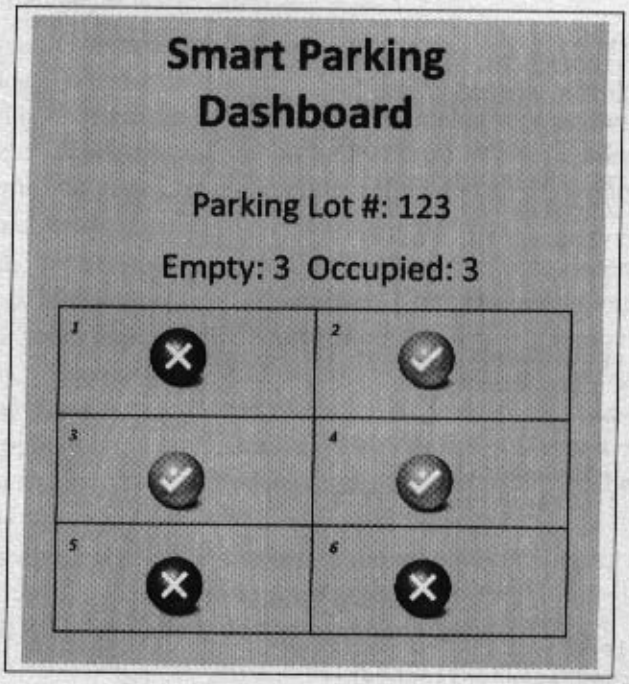


Figure 9.25: Samrt Parking web application screenshot

A Python implementation of the controller service native service that runs on Raspberry Pi, is shown in Box 9.18. The *runController* function is called every second and the reading of the ultrasonic sensor is obtained. If the distance returned by the sensor is less than a threshold, the slot is considered to be occupied. The current state of the slot is then updated by sending a PUT request to the state REST service.

Box 9.18: Python code for controller native service - smart parking - controller.py

```
import RPi.GPIO as GPIO
import time
import sys

GPIO.setmode(GPIO.BCM)
SENSOR_PIN = 27
TRIGGER_PIN=17
threshold = 10000

def readUltrasonicSensor():
   GPIO.setup(TRIGGER_PIN,GPIO.OUT)
   GPIO.setup(SENSOR_PIN,GPIO.IN)
   GPIO.output(TRIGGER_PIN, GPIO.LOW)
   time.sleep(0.3)
   GPIO.output(TRIGGER_PIN, True)
   time.sleep(0.00001)
   GPIO.output(TRIGGER_PIN, False)
   while GPIO.input(SENSOR_PIN) == 0:
      signaloff = time.time()

   while GPIO.input(SENSOR_PIN) == 1:
      signalon = time.time()

   timepassed = signalon - signaloff
   distance = timepassed * 17000
   if distance < threshold:
      return 1
   else:
      return 0

def runController():
   pinState = readUltrasonicSensor()
   if pinState == 1:
      setCurrentState('occupied')
   else:
      setCurrentState('empty')

def setCurrentState(val):
   values = {"name":  val}
   r=requests.put('http://127.0.0.1:8000/state/1/',
    data=values, auth=('username', 'password'))
```

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

9.4 Environment

9.4.1 Weather Monitoring System

REST-based Implementation

A design of a weather monitoring IoT system was described in Chapter-5 using the IoT design methodology. A concrete implementation of the system based on Django framework is described in this section. The purpose of the weather monitoring system is to collect data on environmental conditions such as temperature, pressure, humidity and light in an area using multiple end nodes. The end nodes send the data to the cloud where the data is aggregated and analyzed.

Figure 9.26 shows the deployment design for the system. The system consists of multiple nodes placed in different locations for monitoring temperature, humidity and pressure in an area. The end nodes are equipped with various sensors (such as temperature, pressure, humidity and light). The end nodes send the data to the cloud and the data is stored in a cloud database. The analysis of data is done in the cloud to aggregate the data and make predictions. A cloud-based application is used for visualizing the data. The centralized controller can send control commands to the end nodes, for example, to configure the monitoring interval on the end nodes.

Figure 9.27 shows a schematic diagram of the weather monitoring system. The devices and components used in this example are Raspberry Pi mini computer, temperature and humidity sensor (DHT22), pressure and temperature sensor (BMP085) and LDR sensor. An analog-to-digital (A/D) converter (MCP3008) is used for converting the analog input from LDR to digital.

Figure 9.28 shows the specification of the controller service for the weather monitoring system. The controller service runs as a native service on the device and monitors temperature, pressure, humidity and light every 10 seconds. The controller service calls the REST service to store these measurements in the cloud. This example uses the Xively Platform-as-a-Service for storing data. In the *setupController* function, new Xively datastreams are created for temperature, pressure, humidity and light data. The *runController* function is called every 10 seconds and the sensor readings are obtained.

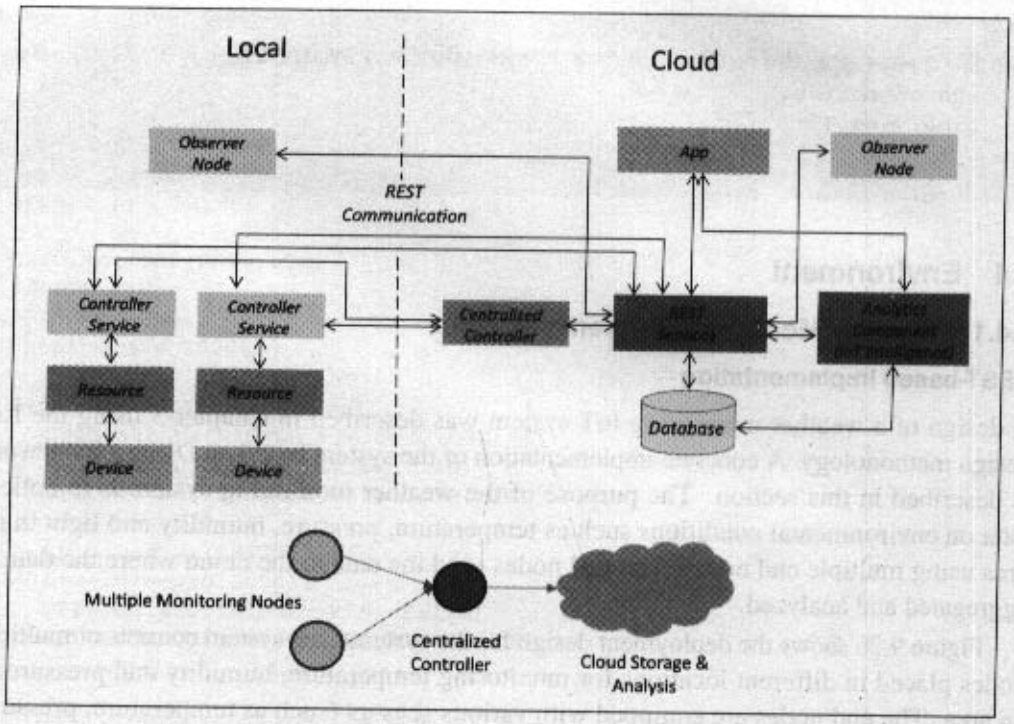


Figure 9.26: Deployment design of the weather monitoring IoT system

Box 9.19: Python code for controller native service - controller.py

```
import time
import datetime
import sys
import json
import requests
import xively
import subprocess
from random import randint
import dhtreader
from Adafruit_BMP085 import BMP085
import spidev

global temp_datastream
global pressure_datastream
```

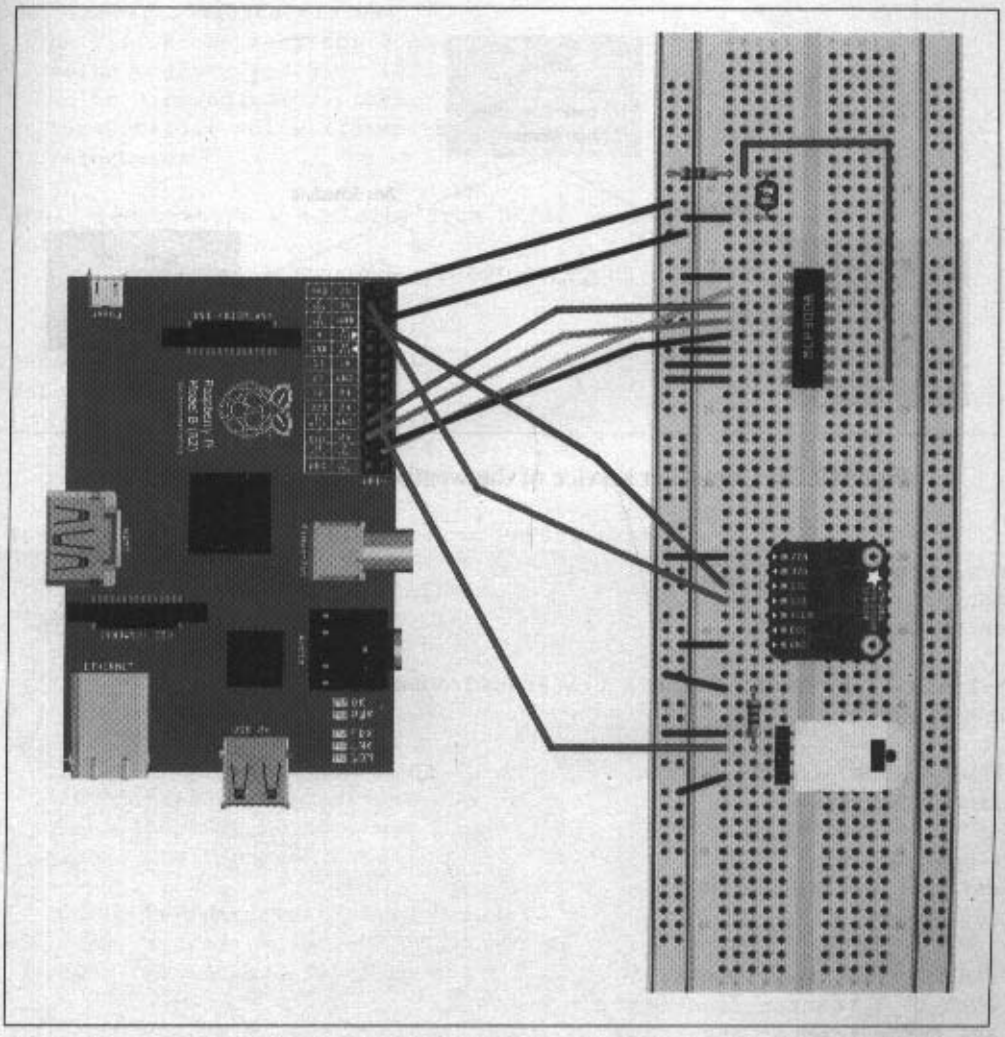


Figure 9.27: Schematic diagram of a weather monitoring end-node showing the device and sensors

```
global humidity_datastream
global light_datastream

#Initialize DHT22 Temperature/Humidity Sensor
```

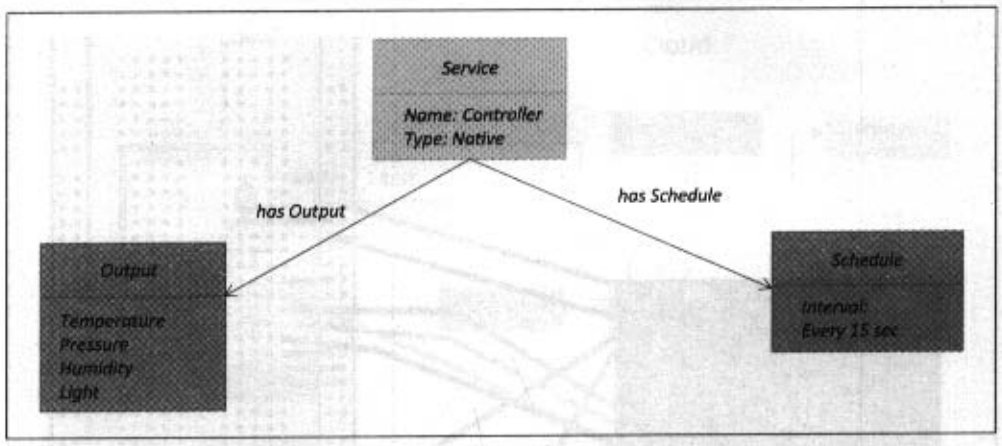


Figure 9.28: Controller service of the weather monitoring IoT system

```
dev_type = 22
dht_pin = 24
dhtreader.init()

# Initialise BMP085 Temperature/Pressure Sensor
bmp = BMP085(0x77)

#LDR channel on MCP3008
light_channel = 0

# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)

#Initialize Xively Feed
FEED_ID = "<enter feed-id>"
API_KEY = "<enter apr-key>"
api = xively.XivelyAPIClient(API_KEY)

# Function to read SPI data from MCP3008 chip
def ReadChannel(channel):
   adc = spi.xfer2([1,(8+channel)<<4,0])
   data = ((adc[1]&3) << 8) + adc[2]
   return data

#Function to convert LDR reading to Lux
```

```
def ConvertLux(data,places):
    R=10 #10k-ohm resistor connected to LDR
    volts = (data * 3.3) / 1023
    volts = round(volts,places)
    lux=500*(3.3-volts)/(R*volts)
    return lux

#Read temperature & humidity from DHT22 sensor
def read_DHT22_Sensor():
    temp, humidity = dhtreader.read(dev_type, dht_pin)
    return temp, humidity

#Read LDR connected to MCP3008
def readLDR():
    light_level = ReadChannel(light_channel)
    lux = ConvertLux(light_level,2)
    return lux

#Read temperature & pressure from BMP085 sensor
def read_BMP085_Sensor():
    temp = bmp.readTemperature()
    pressure = bmp.readPressure()
    return temp, pressure

#Controller main function
def runController():
    global temp_datastream
    global pressure_datastream
    global humidity_datastream
    global light_datastream

    temp1, humidity=read_DHT22_Sensor()
    temp2, pressure=read_BMP085_Sensor()
    temp=(temp1+temp2)/2 #take avg

    light=readLDR()

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

    pressure_datastream.current_value = pressure
    pressure_datastream.at = datetime.datetime.utcnow()

    humidity_datastream.current_value = humidity
    humidity_datastream.at = datetime.datetime.utcnow()
```

```
    light_datastream.current_value = light
    light_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)

    print "Updating Xively feed with Humidity:  %s" % humidity
    try:
       pressure_datastream.update()
    except requests.HTTPError as e:
       print "HTTPError({0}):  {1}".format(e.errno, e.strerror)

    print "Updating Xively feed with Pressure:  %s" % pressure
    try:
       humidity_datastream.update()
    except requests.HTTPError as e:
       print "HTTPError({0}):  {1}".format(e.errno, e.strerror)

    print "Updating Xively feed with Light:  %s" % light
    try:
       light_datastream.update()
    except requests.HTTPError as e:
       print "HTTPError({0}):  {1}".format(e.errno, e.strerror)

#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

#Get existing or create new Xively data stream for pressure
def get_pressuredatastream(feed):
    try:
       datastream = feed.datastreams.get("pressure")
       return datastream
    except:
       datastream = feed.datastreams.create("pressure", tags="pressure")
```

```
        return datastream

#Get existing or create new Xively data stream for humidity
def get_humiditydatastream(feed):
    try:
        datastream = feed.datastreams.get("humidity")
        return datastream
    except:
        datastream = feed.datastreams.create("humidity", tags="humidity")
        return datastream

#Get existing or create new Xively data stream for light
def get_lightdatastream(feed):
    try:
        datastream = feed.datastreams.get("light")
        return datastream
    except:
        datastream = feed.datastreams.create("light", tags="light")
        return datastream

#Controller setup function
def setupController():
    global temp_datastream
    global pressure_datastream
    global humidity_datastream
    global light_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

    pressure_datastream = get_pressuredatastream(feed)
    pressure_datastream.max_value = None
    pressure_datastream.min_value = None

    humidity_datastream = get_humiditydatastream(feed)
    humidity_datastream.max_value = None
    humidity_datastream.min_value = None
```

```
    light_datastream = get_lightdatastream(feed)
    light_datastream.max_value = None
    light_datastream.min_value = None


setupController()
while True:
    runController()
    time.sleep(10)
```

Box 9.20 shows the HTML and JavaScript code for the web page that displays the weather information for a location. The web page uses the Xively JavaScript library [128] to fetch the weather data from the Xively cloud.

■ Box 9.20: Code for a web page for displaying weather information

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<link href="readable.css" rel="stylesheet">
<style>
body {
padding-top:  80px;
}
</style>
<link href="bootstrap-responsive.css" rel="stylesheet">

<!-[if lt IE 9]>
<script src="http://html5shim.googlecode.com/svn/trunk/html5.js">
</script>
<![endif]-> <style type="text/css">
hr {
margin:  0 0;
}
#map_canvas {
width:  100%;
height:  100%;
min-height:  100%;
display:  block;
border-radius:  10px;
-webkit-border-radius:  10px;
```

```
}

.well {
width:  100%;
height:  100%;
min-height:  100%;
}

.alert {
border:  1px solid rgba(229,223,59,0.78);
}
</style>
<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(30.733315,76.779418);
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:"Location"
});
startws ();

}

</script>
<title>Weather Station</title>
</head>
```

```
<body onload="initialize()">
<div class="container">
<div class="row" style="height:20px"> </div>
<div class="row"> <div class="span12">
<center><h1>Weather Station</h1></center>
<br>
<h3>CityName</h3>
</div></div>
<div class="row">
<div class="span6">
<div class='row'>
<div class='span3'><h4>Temperature</h4></div>
<div class='span3'>
<h4 id='temperature'></h4></div></div><hr/>
<div class='row'>
<div class='span3'><h4>Humidity</h4></div>
<div class='span3'> <h4 id='humidity'></h4></div></div><hr/>
<div class='row'>
<div class='span3'><h4>Pressure</h4></div>
<div class='span3'> <h4 id='pressure'></h4></div></div><hr/>
<div class='row'>
<div class='span3'><h4>Light sensor</h4></div>
<div class='span3'> <h4 id='light'></h4></div></div></div>
<div class="span6" style="height:435px">
<b>Location:</b> CityName | <b>Exposure:</b> outdoor |
<b>Disposition:</b> fixed<div class="well">
<div id = "map_canvas">
</div>
</div>
</div>
</div>
</div>
<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
</script>
<script
src="http://d23cj0cdvyoxg0.cloudfront.net/xivelyjs-1.0.4.min.js">
</script>
<script>
$(document).ready(function($) {
xively.setKey( "<enter api key>" );
var feedID = <enter feed-id>, // Feed ID
temp_datastreamID = "temperature";
pressure_datastreamID = "pressure";
humidity_datastreamID = "humidity";
```

```
light_datastreamID = "light";

temp_selector = "#temperature";
pressure_selector = "#pressure";
humidity_selector = "#humidity";
light_selector = "#light";

// Get datastream data from Xively
xively.datastream.get (feedID, temp_datastreamID, function(datastream) {
// Display the current value from the datastream
$(temp_selector).html( datastream["current_value"].concat(" C") );
});

xively.datastream.get (feedID,
pressure_datastreamID, function ( datastream ) {
$(pressure_selector).html( datastream["current_value"].concat(" mb") );
});

xively.datastream.get (feedID,
humidity_datastreamID, function ( datastream ) {
$(humidity_selector).html( datastream["current_value"].concat(" %") );
});

xively.datastream.get (feedID,
light_datastreamID, function ( datastream ) {
$(light_selector).html( datastream["current_value"].concat(" L") );
});

});
</script>

</body>
</html>
```

An alternative to using the Xively JavaScript API, is to use the Xively Python library with a Django application. Box 9.21 shows the code for a Django view that retrieves data from the Xively cloud.

■ Box 9.21: Code for a 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")
pressure_datastream = feed.datastreams.get("pressure")
humidity_datastream = feed.datastreams.get("humidity")
light_datastream = feed.datastreams.get("light")

def home(request):
   temperature=temp_datastream.current_value
   pressure=pressure_datastream.current_value
   humidity=humidity_datastream.current_value
   light=light_datastream.current_value
   lat=feed.location.lat
   lon=feed.location.lon

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

Figure 9.29 shows a screenshot of the weather monitoring web application.

WebSocket-based Implementation

The previous section described a REST-based implementation of the weather monitoring IoT system. In this section you will learn about an alternative implementation of the weather monitoring IoT system based on WebSocket.

The WebSocket implementation is based the Web Application Messaging Protocol (WAMP) which is a sub-protocol of WebSocket. You learned about Autobahn, an open source implementation of WAMP in Chapter 8. The deployment design for the WebSocket implementation is the same as shown in Figure 9.26.

Figure 9.30 shows the communication between various components of the WebSocket implementation. The controller in the WebSocket implementation is a WAMP application component that runs over a WebSocket transport client on the IoT device. WAMP application router runs on a WebSocket transport server on the server instance in the cloud. The role of the Client on the device in this example is that of a Publisher, while the role of the Router is that of a Broker. Publisher publishes messages to the topics managed by the Broker. Subscribers subscribe to topics they are interested in with Brokers. Brokers route events

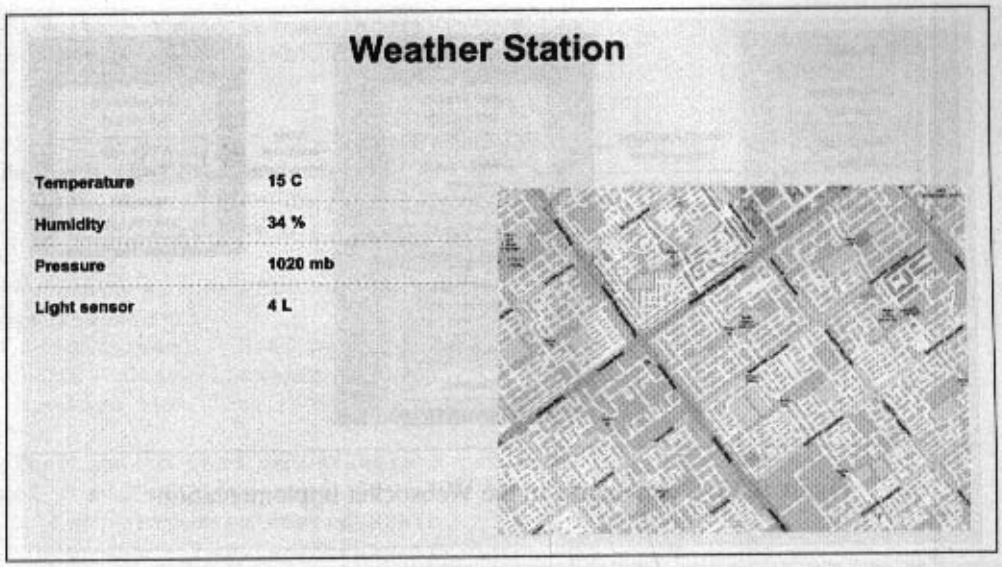


Figure 9.29: Screenshot of weather monitoring web application

incoming from Publishers to Subscribers that are subscribed to respective topics. Brokers decouple the Publisher and Subscriber. In this example, the Publisher and Subscriber run application code. The Publisher application component is the controller component, the source code of which is shown in Box 9.22. The Subscriber application component is the web frontend, the source code of which is shown in Box 9.23. The analytics component runs on a separate instance and subscribes to the topics managed by the Broker. Box 9.25 shows the code for a dummy analytics component. The communication between Publisher - Broker and Broker - Subscribers happens over a WAMP-WebSocket session.

Box 9.22: Controller code for WebSocket implementation - weathercontroller.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 dhtreader
from Adafruit_BMP085 import BMP085
import spidev
```

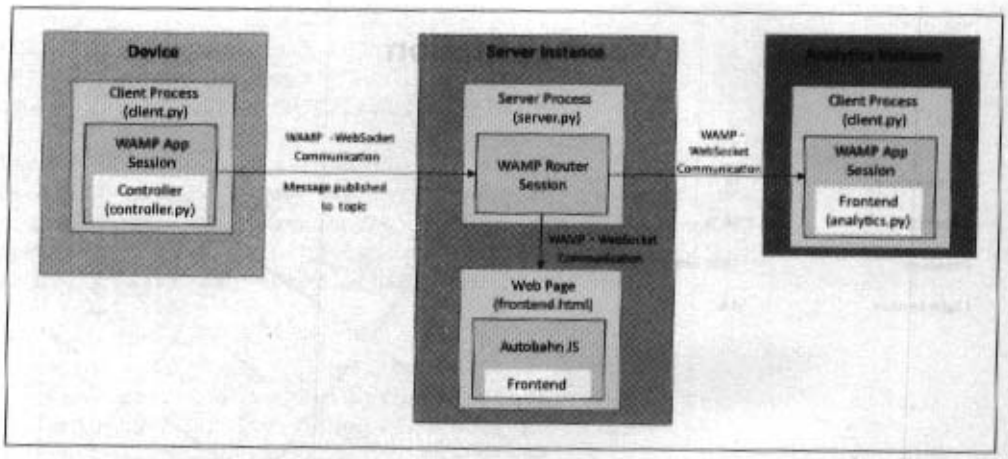


Figure 9.30: Components in the Websocket implementation

```
#Initialize DHT22 Temperature/Humidity Sensor
dev_type = 22
dht_pin = 24
dhtreader.init()

# Initialise BMP085 Temperature/Pressure Sensor
bmp = BMP085(0x77)

#LDR channel on MCP3008
light_channel = 0

# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)

# Function to read SPI data from MCP3008 chip
def ReadChannel(channel):
   adc = spi.xfer2([1,(8+channel)<<4,0])
   data = ((adc[1]&3) << 8) + adc[2]
   return data

#Function to convert LDR reading to Lux
def ConvertLux(data,places):
   R=10 #10k-ohm resistor connected to LDR
   volts = (data * 3.3) / 1023
```

```
    volts = round(volts,places)
    lux=500*(3.3-volts)/(R*volts)
    return lux

#Function to read temperature & humidity from DHT22 sensor
def read_DHT22_Sensor():
    temp, humidity = dhtreader.read(dev_type, dht_pin)
    return temp, humidity

#Function to read LDR connected to MCP3008
def readLDR():
    light_level = ReadChannel(light_channel)
    lux = ConvertLux(light_level,2)
    return lux

#Function to read temperature & pressure from BMP085 sensor
def read_BMP085_Sensor():
    temp = bmp.readTemperature()
    pressure = bmp.readPressure()
    return temp, pressure

#Controller main function
def runController():
    temp1, humidity=read_DHT22_Sensor()
    temp2, pressure=read_BMP085_Sensor()
    temperature=(temp1+temp2)/2 #take avg
    light=readLDR()

    datalist = [temperature, humidity, pressure, light]

    return datalist


#An application component that publishes sensor data every second.
class Component(ApplicationSession):
    @inlineCallbacks
    def onJoin(self, details):
        while True:
            datalist = runcontroller()
            self.publish('com.myapp.topic1', datalist)
            yield sleep(1)
```

■ Box 9.23: Code for a web page for displaying weather information - WebSocket implementation - frontend.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Weather Station</title>
<!- Le styles ->
<link href="readable.css" rel="stylesheet">
<style>
body {
padding-top: 80px;
}
</style>
<link href="bootstrap-responsive.css" rel="stylesheet">
<!- HTML5 shim, for IE6-8 support of HTML5 elements ->
<!-[if lt IE 9]>
<script
src="http://html5shim.googlecode.com/svn/trunk/html5.js">
</script>
<![endif]-> <style type="text/css">
hr {
margin: 0 0;
}
#map_canvas {
width: 100%;
height: 100%;
min-height: 100%;
display: block;
border-radius: 10px;
-webkit-border-radius: 10px;
}

.well {
width: 100%;
height: 100%;
min-height: 100%;
}

.alert {
border: 1px solid rgba(229,223,59,0.78);
}
```

```
</style>
<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(30.733315,76.779418);
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, Country"
});

}
</script>
<script
src="https://autobahn.s3.amazonaws.com/autobahnjs/
latest/autobahn.min.jgz"> </script>
</head>
<body onload="initialize()">
<div class="container">
<div class="row" style="height:20px"> </div>
<div class="row"> <div class="span12">
<center><h1>Weather Station</h1></center>
<br>
<h3>CityName</h3>
</div></div>
<div class="row">
<div class="span6">
<div class='row'>
```

```
<div class='span3'><h4>Temperature</h4></div>
<div class='span3'> <h4 id='temperature'></h4>
</div></div><hr/><div class='row'>
<div class='span3'><h4>Humidity</h4></div>
<div class='span3'> <h4 id='humidity'></h4></div>
</div><hr/><div class='row'>
<div class='span3'><h4>Pressure</h4></div>
<div class='span3'> <h4 id='pressure'></h4></div>
</div><hr/><div class='row'>
<div class='span3'><h4>Light sensor</h4></div>
<div class='span3'> <h4 id='light'></h4></div></div></div>
<div class="span6" style="height:435px">
<b>Location:</b> CityName, Country | <b>
Exposure:</b> outdoor | <b>Disposition:</b> fixed
<div class="well">
<div id = "map_canvas">
</div>
</div>

<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js">
</script>

<script>
try {
var autobahn = require('autobahn');
} catch (e) {
// when running in browser, AutobahnJS will
// be included without a module system
}

var connection = new autobahn.Connection({
url:  'ws://127.0.0.1:8080/ws',
realm:  'realm1'}
);

connection.onopen = function (session) {

var received = 0;
temp_selector = "#temperature";
pressure_selector = "#pressure";
humidity_selector = "#humidity";
light_selector = "#light";

function onevent1(args) {
```

```
//console.log("Got event:", args);

$(temp_selector).html( args[0][0].concat(" C") );
$(pressure_selector).html( args[0][1].concat(" mb") );
$(humidity_selector).html( args[0][2].concat(" %") );
$(light_selector).html( args[0][3].concat(" L") );
}

session.subscribe('com.myapp.topic1', onevent1);
};
connection.open();
</script>
</body>
</html>
```

■ Box 9.24: Commands for running WebSocket implementation of weather monitoring example

```
#Setup Autobahn-Python
sudo apt-get install python-twisted python-dev
sudo apt-get install python-pip
sudo pip install -upgrade twisted
sudo pip install -upgrade autobahn

#Clone AutobahnPython
git clone https://github.com/tavendo/AutobahnPython.git

#Create weathercontroller.py as shown in Box 9.22

#Run the application router on a WebSocket transport server

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

#Run controller component over a WebSocket transport client
python AutobahnPython/examples/twisted/wamp/basic/client.py -component
"weathercontroller.Component"

#Create frontend.html as shown in Box 9.23
#Open frontend.html in a Browser
#Sensor readings would be updated in the web page shown in Figure 9.29
```

■ Box 9.25: Code for a dummy analytics component - WebSocket implementation - analytics.py

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

#Placeholder for analysis function
def analyzeData(data):
   return true

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

   self.received = 0

   def on_event(data):
      print "Temperature:  " + data[0] + "; Humidity:  " + data[1]
      + "; Pressure:  " + data[2] + "; Light:  " + data[3]


   #Placeholder for analysis function
   analyzeData(data)

   yield self.subscribe(on_event, 'com.myapp.topic1')


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

9.4.2 Weather Reporting Bot

This case study is about a weather reporting bot which reports weather information by sending tweets on Twitter. Figure 9.31 shows a schematic of the weather monitoring end-node. The end-node is comprised of a Raspberry Pi mini-computer, temperature, humidity, pressure and light sensors. In addition to the sensors, a USB webcam is also attached to the device.

Box 9.26 shows the Python code for the controller service that runs on the end-node. The controller service obtains the temperature, humidity, pressure and light readings from the sensors, every 30 minutes. At at the same time an image is captured from the webcam attached to the device. The sensor readings and the captured image is then sent as a tweet on Twitter. To send tweets the controller service uses a Python library for Twitter called *tweepy*.

With tweepy you can use the Twitter REST API to send tweets. Before using the Twitter API, you would need to setup a Twitter developer account and then create a new application (with read-write permissions). Upon creating the application you will get the API key, API secret and access tokens. These credentials and tokens are used in the controller service.

▪ Box 9.26: Python code for weather reporting bot that tweets weather updates to Twitter

```
import time
import datetime
import sys
from random import randint
import dhtreader
from Adafruit_BMP085 import BMP085
import spidev
from SimpleCV import Camera
from time import sleep
import tweepy


#Initialize USB webcam
myCamera = Camera(prop_set={'width':320, 'height': 240})

#Twitter Application Credentials
CONSUMER_KEY ="<enter>"
CONSUMER_SECRET = "<enter>"
ACCESS_KEY = "<enter>"
ACCESS_SECRET = "<enter>"

auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
auth.set_access_token(ACCESS_KEY, ACCESS_SECRET)
api = tweepy.API(auth)


#Initialize DHT22 Temperature/Humidity Sensor
dev_type = 22
dht_pin = 24
dhtreader.init()

# Initialise BMP085 Temperature/Pressure Sensor
bmp = BMP085(0x77)

#LDR channel on MCP3008
```

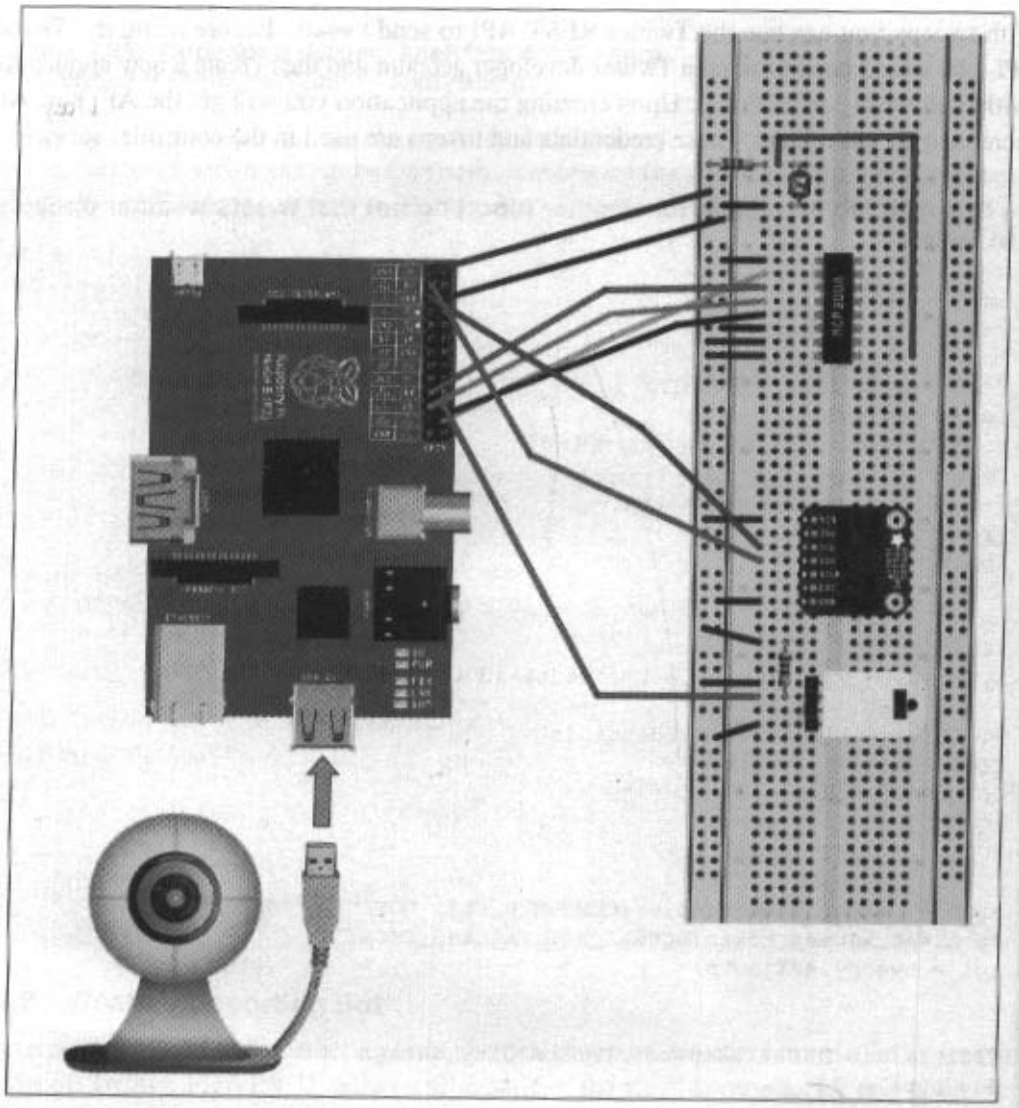


Figure 9.31: Schematic diagram of a weather reporting bot end-node showing the device and sensors

```
light_channel = 0
```

```
# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)

#Initialize Xively Feed
FEED_ID = "<enter>"
API_KEY = "<enter>"
api = xively.XivelyAPIClient(API_KEY)

# Function to read SPI data from MCP3008 chip
def ReadChannel(channel):
   adc = spi.xfer2([1,(8+channel)<<4,0])
   data = ((adc[1]&3) << 8) + adc[2]
   return data

#Function to convert LDR reading to Lux
def ConvertLux(data,places):
   R=10 #10k-ohm resistor connected to LDR
   volts = (data * 3.3) / 1023
   volts = round(volts,places)
   lux=500*(3.3-volts)/(R*volts)
   return lux

#Function to read temperature & humidity from DHT22 sensor
def read_DHT22_Sensor():
   temp, humidity = dhtreader.read(dev_type, dht_pin)
   return temp, humidity

#Function to read LDR connected to MCP3008
def readLDR():
   light_level = ReadChannel(light_channel)
   lux = ConvertLux(light_level,2)
   return lux

#Function to read temperature & pressure from BMP085 sensor
def read_BMP085_Sensor():
   temp = bmp.readTemperature()
   pressure = bmp.readPressure()
   return temp, pressure

#Controller main function
def runController():

   #Get sensor readings
   temp1, humidity=read_DHT22_Sensor()
```

```
    temp2, pressure=read_BMP085_Sensor()
    temperature=(temp1+temp2)/2 #take avg
    light=readLDR()

    #Capture Image
    frame = myCamera.getImage()
    frame.save("weather.jpg")

    status = "Weather Update at:  " +
    datetime.datetime.now().strftime('%Y/%m/%d %H:%M:%S') +
       " - Temperature:  "+ temperature + ", Humidity:  "+
        humidity + ", Pressure:  "+ pressure + ", Light:  "+ light

    photo_path = '/home/pi/weather.jpg'

    #Tweet weather information with photo
    api.update_with_media(photo_path, status=status)


setupController()
while True:
    runController()
    time.sleep(1800)
```

Figure 9.32 shows a screenshot of a tweet sent by the weather reporting bot on Twitter.

9.4.3 Air Pollution Monitoring

IoT based air pollution monitoring systems can monitor emission of harmful gases by factories and automobiles using gaseous and meteorological sensors. This section provides an implementation of an air pollution monitoring IoT system. The deployment design for the system is similar to the deployment shown in Figure 9.26. The system design steps are similar to the weather monitoring system described in the previous section. Therefore only the schematic design and the controller implementation is provided.

The system consists of multiple nodes placed in different locations for monitoring air pollution in an area. The end nodes are equipped with CO and NO_2 sensors. The end nodes send the data to the cloud and the data is stored in a cloud database. A cloud-based application is used for visualizing the data.

Figure 9.33 shows a schematic diagram of air pollution monitoring end-node. The end node includes a Raspberry Pi mini-computer, MICS-2710 NO_2 sensor and MICS-5525 CO

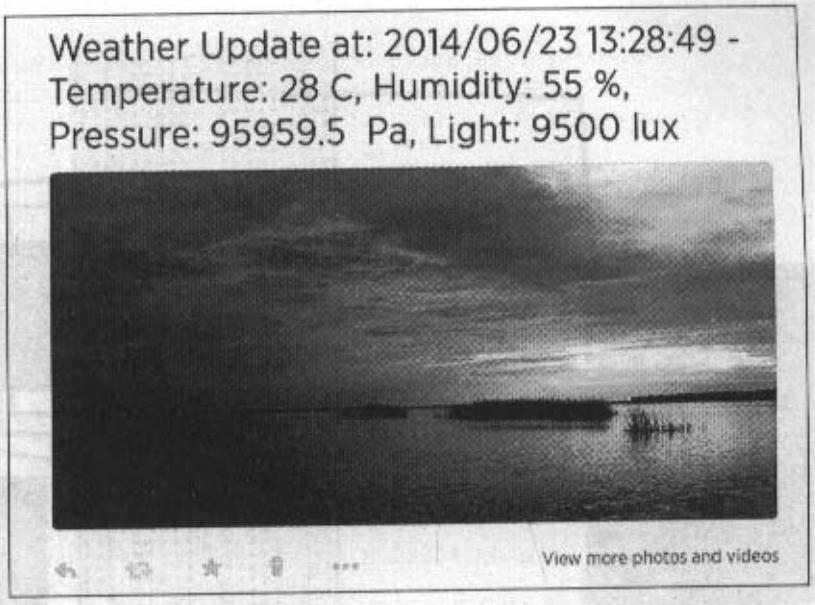


Figure 9.32: Screenshot of a weather update tweeted by the weather reporting bot

sensor. An A/D converter (MCP3008) is used for converting the analog inputs from the sensors to digital.

Box 9.27 shows the implementation of the native controller service for air pollution monitoring system. The controller service runs as a native service on the device and obtains the sensor readings every 10 seconds. The controller service calls the Xively REST service to store these measurements in the cloud.

Box 9.27: Python code for controller native service - air pollution monitoring system

```
import time
import datetime
import sys
import json
import requests
import xively
import spidev

global NO2_datastream
```

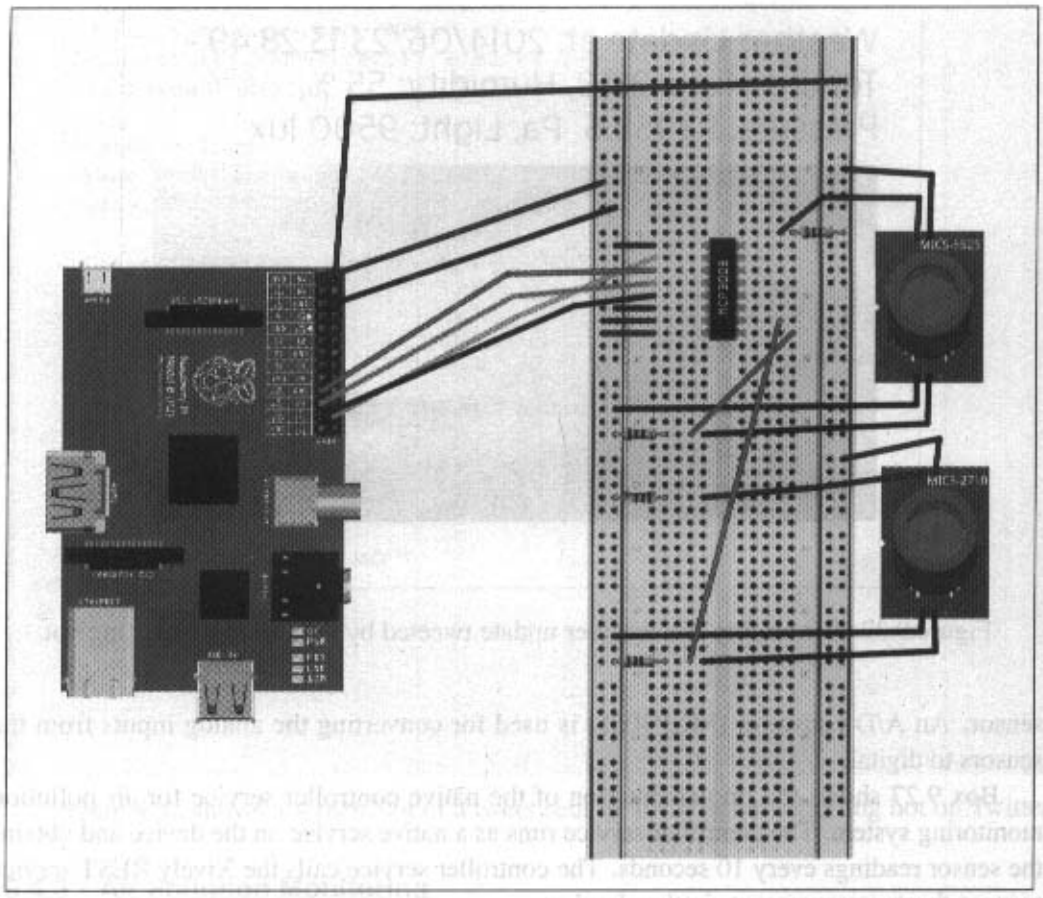


Figure 9.33: Schematic diagram of air pollution monitoring end-node showing the device and sensors

```
global CO_datastream

#Sensor channel on MCP3008
CO_CHANNEL = 0
NO2_CHANNEL = 1

vin=5
r0=10000
pullup = 10000
```

```
#Conversions based on Rs/Ro vs ppm plots of the sensors
CO_Conversions = [((0,100),(0,0.25)),((100,133),(0.25,0.325)),
((133,167),(0.325,0.475)),((167,200),(0.475,0.575)),((200,233),
(0.575,0.665)),((233,267),(0.666,0.75))]
NO2_Conversions = [((0,100),(0,0.25)),((100,133),(0.25,0.325)),
((133,167),(0.325,0.475)),((167,200),(0.475,0.575)),((200,233),
(0.575,0.665)),((233,267),(0.666,0.75))]

# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)

#Initialize Xively Feed
FEED_ID = "467475686"
API_KEY = "OzMuaKpacvlDNgOrXl6SA3WNb9n83BfT51MfEEkLVHHZiEDB"
api = xively.XivelyAPIClient(API_KEY)

# Function to read SPI data from MCP3008 chip
def ReadChannel(channel):
   adc = spi.xfer2([1,(8+channel)<<4,0])
   data = ((adc[1]&3) << 8) + adc[2]
   return data

def get_resistance(channel):
   result = ReadChannel(channel)
   if result == 0:
      resistance = 0
   else:
      resistance = (vin/result - 1)*pullup
   return resistance

def converttoppm(rs,conversions):
   rsper = 100*(float(rs)/r0)
   for a in conversions:
        if a[0][0]>=rsper>a[0][1]:
             mid, hi = rsper-a[0][0],a[0][1]-a[0][0]
             sf = float(mid)/hi
             ppm = sf * (a[1][1]-a[1][0]) + a[1][0]
              return ppm
   return 0

def get_NO2():
   rs = get_resistance(NO2_CHANNEL)
   ppm = converttoppm(rs,NO2_Conversions)
```

```
    return ppm

def get_CO():
    rs = get_resistance(CO_CHANNEL)
    ppm = converttoppm(rs,CO_Conversions)
    return ppm

#Controller main function
def runController():
    global NO2_datastream
    global CO_datastream

    NO2_reading=get_NO2()
    CO_reading=get_CO()

    NO2_datastream.current_value = NO2_reading
    NO2_datastream.at = datetime.datetime.utcnow()

    CO_datastream.current_value = CO_reading
    CO_datastream.at = datetime.datetime.utcnow()


    print "Updating Xively feed with CO: %s" % CO
    try:
        CO_datastream.update()
    except requests.HTTPError as e:
        print "HTTPError({0}): {1}".format(e.errno, e.strerror)

    print "Updating Xively feed with NO2: %s" % NO2
    try:
        NO2_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 NO2
def get_NO2datastream(feed):
    try:
        datastream = feed.datastreams.get("NO2")
        return datastream
    except:
        datastream = feed.datastreams.create("NO2", tags="NO2")
        return datastream

#Function to get existing or create new Xively data stream for CO
```

```
def get_COdatastream(feed):
    try:
        datastream = feed.datastreams.get("CO")
        return datastream
    except:
        datastream = feed.datastreams.create("CO", tags="CO")
        return datastream

#Controller setup function
def setupController():
    global NO2_datastream
    global CO_datastream

    feed = api.feeds.get(FEED_ID)

    feed.location.lat="30.733315"
    feed.location.lon="76.779418"
    feed.tags="Pollution"
    feed.update()

    NO2_datastream = get_NO2datastream(feed)
    NO2_datastream.max_value = None
    NO2_datastream.min_value = None

    CO_datastream = get_COdatastream(feed)
    CO_datastream.max_value = None
    CO_datastream.min_value = None


setupController()
while True:
    runController()
    time.sleep(10)
```

9.4.4 Forest Fire Detection

IoT based forest fire detection systems use a number of monitoring nodes deployed at different locations in a forest. Each monitoring node collects measurements on ambient conditions (such as temperature and humidity) to predict whether a fire has broken out.

An implementation of a forest fire detection system is described in this section. The system is based on a level-5 IoT deployment with multiple end nodes and one coordinator node. The end nodes perform sensing and the coordinator node collects data from the end

nodes and sends to the cloud.

Figure 9.34 shows a schematic diagram of forest fire detection end-node. The end node includes a Raspberry Pi mini-computer and DHT-22 temperature and humidity sensor. An XBee module is used for wireless communication between the end-node and the coordinator node. Figure 9.35 shows a schematic diagram of the coordinator node.

Boxes 9.28 and 9.29 show the implementations of the native controller services for the end node and coordinator node respectively. The controller service on the end node obtains the sensor readings every 10 seconds and writes the data to the XBee module which sends the data to the coordinator node. The controller service on the coordinator node receives the data from all end nodes and calls the Xively REST service to store these measurements in the cloud.

The XBee modules can be configured to communicate with each other using a Windows based application called X-CTU [127]. All XBee modules should have the same network ID and channel. The XBee module for the coordinator node has to be configured as a coordinator and the rest of the modules have to be configured as end devices or routers.

■ Box 9.28: Python code for controller service on end-node - forest fire detection system

```
import time
import datetime
import serial
import dhtreader

#Set Router ID
RouteID='123'

#Initialize DHT22 Temperature/Humidity Sensor
dev_type = 22
dht_pin = 24
dhtreader.init()

#Function to read temperature & humidity from DHT22 sensor
def read_DHT22_Sensor():
   temp, humidity = dhtreader.read(dev_type, dht_pin)
   return temp, humidity


def write_xbee(data):
   xbee=serial.Serial(port='/dev/ttyAMA0',baudrate='9600')
   xbee.write(data)
```

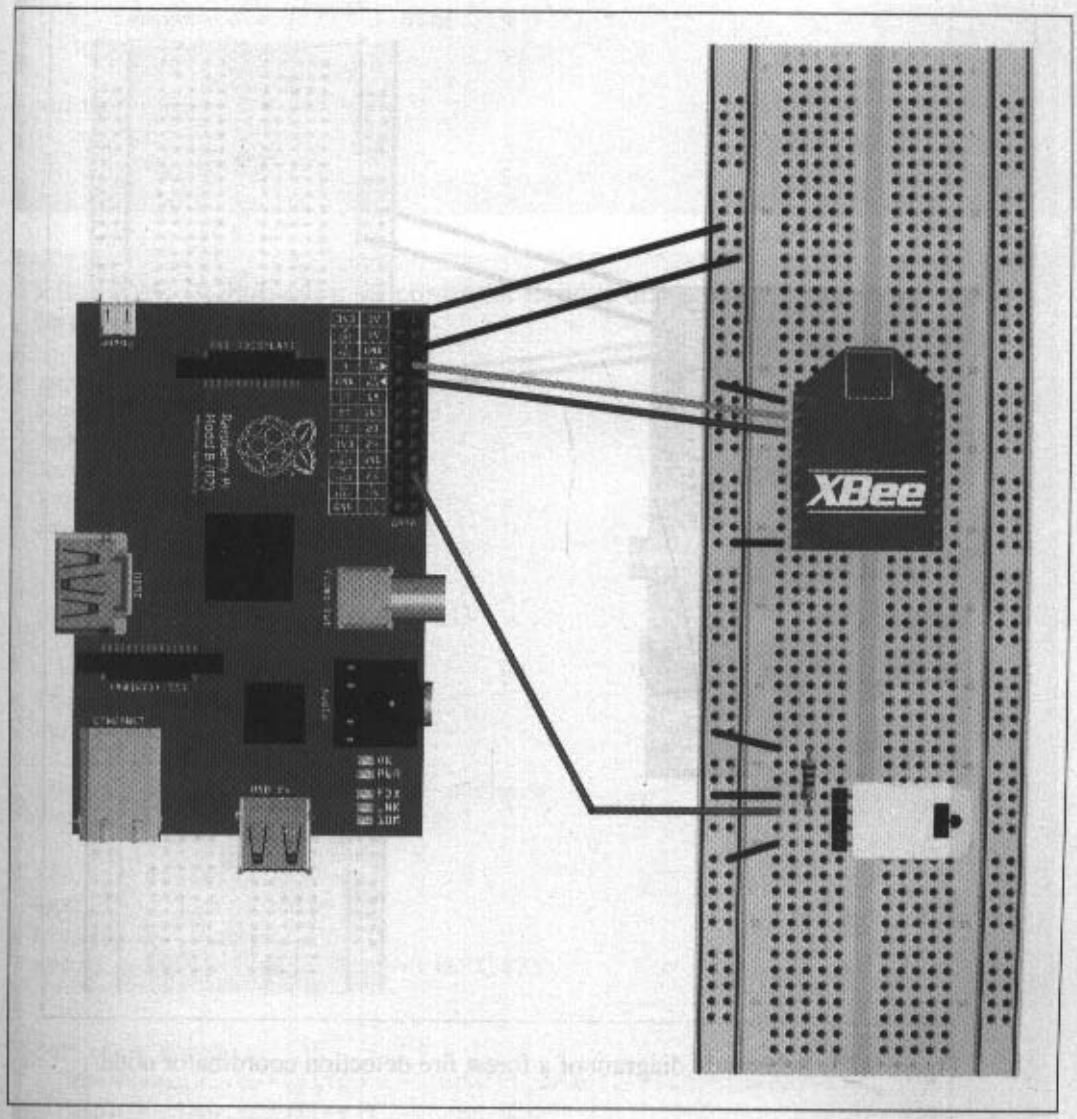


Figure 9.34: Schematic diagram of a forest fire detection end-node showing the device and sensor

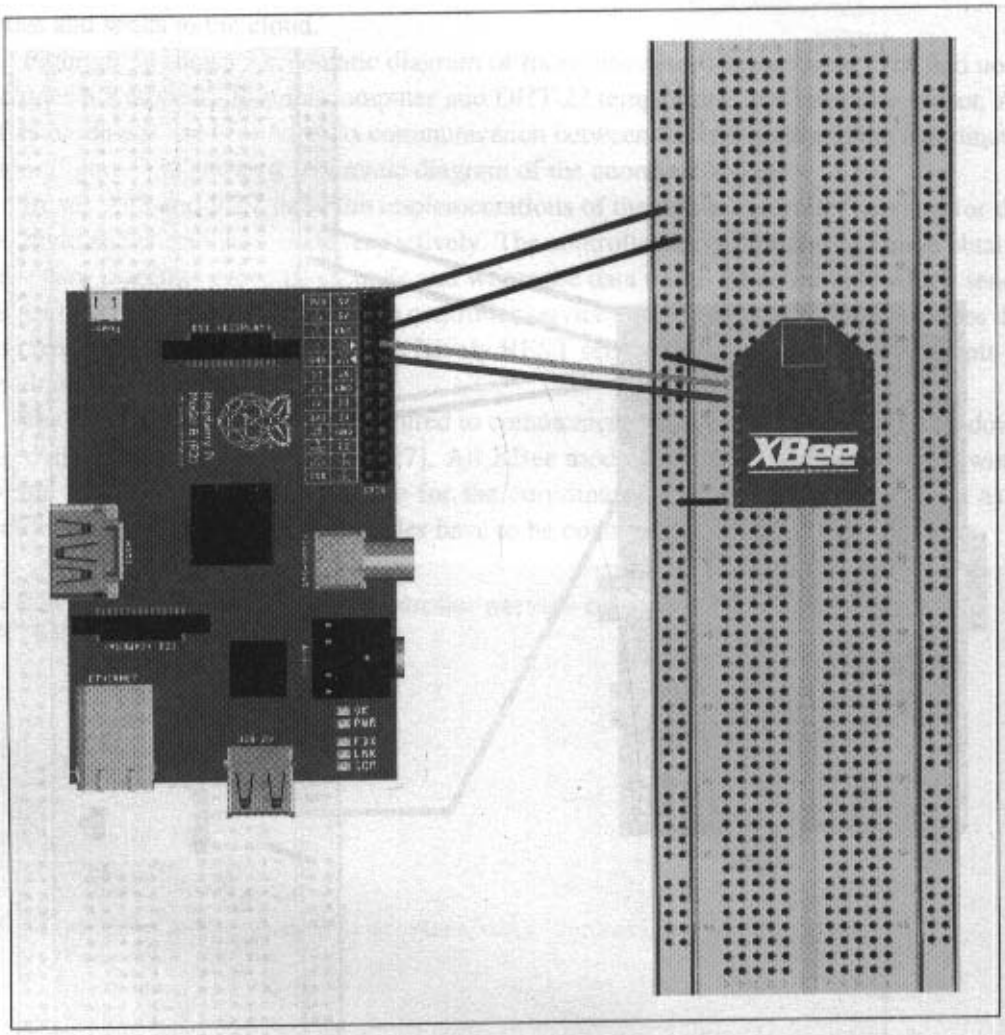


Figure 9.35: Schematic diagram of a forest fire detection coordinator node

```
#Controller main function
def runController():
   temperature, humidity=read_DHT22_Sensor()
   timestamp = str(datetime.datetime.utcnow())

   data= RouteID + "|"+ timestamp + "|"+
```

```
        temperature + "|" + humidity + "| "
    write_xbee(data)

while True:
    runController()
    time.sleep(10)
```

■ Box 9.29: Python code for controller service on coordinator - forest fire detection system

```
import time
import datetime
import sys
import json
import requests
import xively

global temp_datastream = []
global humidity_datastream = []


#Set number of routers
numRouters=2

#Map router ID's to sequence numbers
routerIDs={'123':'1','345':'2'}

#Initialize Xively Feed
FEED_ID = "enter feed id"
API_KEY = "enter key"
api = xively.XivelyAPIClient(API_KEY)


def read_xbee():
    xbee=serial.Serial(port='/dev/ttyAMA0', 9600,timeout=1)
    data = xbee.readline()
    return data

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

```
    data=read_xbee()
    dataArr=data.split('|')

    id = dataArr[0]
    i = routerIDs[id]
    timestamp = dataArr[1]
    temperature = dataArr[2]
    humidity = dataArr[3]

    temp_datastream[i].current_value = temperature
    temp_datastream[i].at = timestamp

    humidity_datastream[i].current_value = humidity
    humidity_datastream[i].at = timestamp

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

    print "Updating Xively feed with Humidity:  %s" % humidity
    try:
       pressure_datastream[i].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
def get_tempdatastream(feed,id):
    try:
       datastream = feed.datastreams.get("temperature"+str(id))
       return datastream
    except:
       datastream = feed.datastreams.create("temperature"+str(id),
tags="temperature")
       return datastream

#Function to get existing or create new Xively data stream for humidity
def get_humiditydatastream(feed,id):
    try:
       datastream = feed.datastreams.get("humidity"+str(id))
       return datastream
    except:
```

```
        datastream = feed.datastreams.create(
            "humidity"+str(id), tags="humidity")
        return datastream

#Controller setup function
def setupController():
    global temp_datastream
    global humidity_datastream

    feed = api.feeds.get(FEED_ID)

    feed.tags="Weather"
    feed.update()

    for i in range(1,numRouters+1):
        temp_datastream[i] = get_tempdatastream(feed,i)
        temp_datastream[i].max_value = None
        temp_datastream[i].min_value = None

        humidity_datastream[i] = get_humiditydatastream(feed,i)
        humidity_datastream[i].max_value = None
        humidity_datastream[i].min_value = None


setupController()
while True:
    runController()
    time.sleep(10)
```

9.5 Agriculture

9.5.1 Smart Irrigation

Smart irrigation systems use IoT devices and soil moisture sensors to determine the amount of moisture in the soil and release the flow of water through the irrigation pipes only when the moisture levels go below a predefined threshold. Data on the moisture levels is also collected in the cloud where it is analyzed to plan watering schedules.

An implementation of a smart irrigation system is described in this section. The deployment design for the system is similar to the deployment shown in Figure 9.26. The system consists of multiple nodes placed in different locations for monitoring soil moisture in a field. The end nodes send the data to the cloud and the data is stored in a cloud database. A cloud-based application is used for visualizing the data. Figure 9.36 shows a

schematic diagram of smart irrigation system end-node. The end node includes a Raspberry Pi mini-computer and soil moisture sensor. A solenoid valve is used to control the flow of water through the irrigation pipe. When the moisture level goes below a threshold, the valve is opened to release water. Box 9.30 shows the Python code for the controller native service for the smart irrigation system.

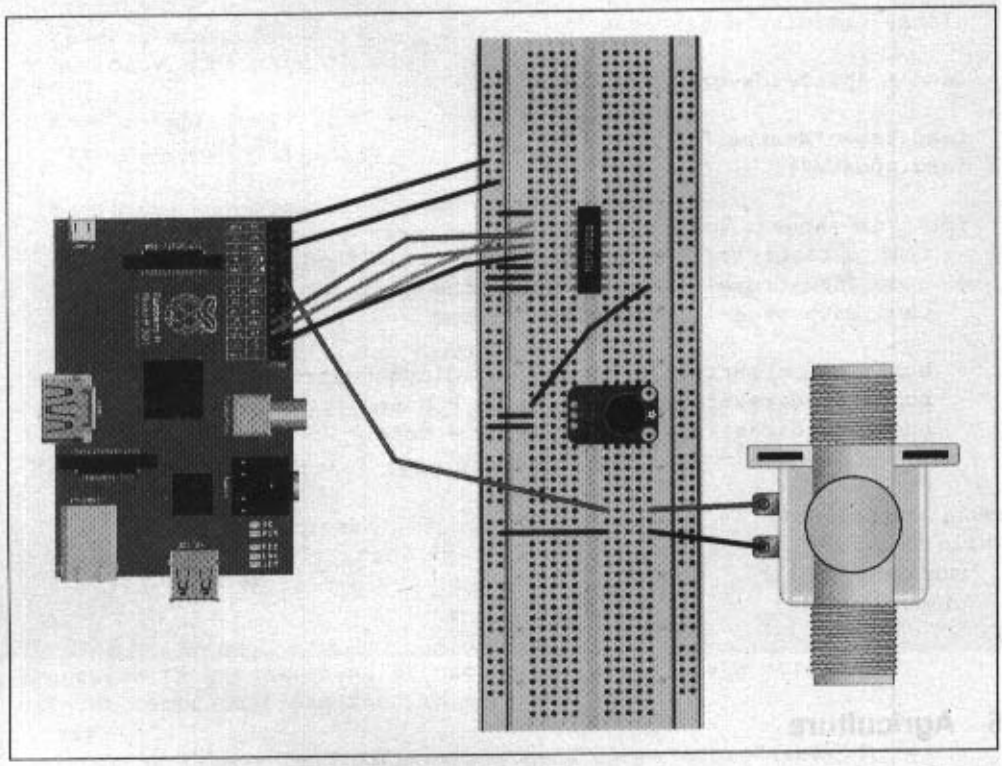


Figure 9.36: Schematic diagram of a smart irrigation system end-node showing the device and sensor

▪ Box 9.30: Python code for controller native service - smart irrigation system

```
import time
import datetime
import sys
```

```
import json
import requests
import xively
import subprocess
import spidev

global mositure_datastream
#LDR channel on MCP3008
mositure_channel = 0

GPIO.setmode(GPIO.BCM)
TRIGGER_PIN=18
threshold = 10

# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)

#Initialize Xively Feed
FEED_ID = "enter feed id"
API_KEY = "enter key"
api = xively.XivelyAPIClient(API_KEY)

# Function to read SPI data from MCP3008 chip
def ReadChannel(channel):
   adc = spi.xfer2([1,(8+channel)<<4,0])
   data = ((adc[1]&3) << 8) + adc[2]
   return data

#Function to read sensor connected to MCP3008
def readMositure():
   level = ReadChannel(mositure_channel)
   return level


#Controller main function
def runController():
   global mositure_datastream

   level=readMositure()

   #Check moisture level
   if (level<threshold):
      GPIO.output(TRIGGER_PIN, True)
   else:
```

```
        GPIO.output(TRIGGER_PIN, False)

    mositure_datastream.current_value = level
    mositure_datastream.at = datetime.datetime.utcnow()

    print "Updating Xively feed with mositure:  %s" % mositure
    try:
        mositure_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
def get_mosituredatastream(feed):
    try:
        datastream = feed.datastreams.get("mositure")
        return datastream
    except:
        datastream = feed.datastreams.create("mositure", tags="mositure")
        return datastream


#Controller setup function
def setupController():
    global mositure_datastream

    feed = api.feeds.get(FEED_ID)

    feed.location.lat="30.733315"
    feed.location.lon="76.779418"
    feed.tags="Soil Moisture"
    feed.update()

    mositure_datastream = get_mosituredatastream(feed)
    mositure_datastream.max_value = None
    mositure_datastream.min_value = None

setupController()
while True:
    runController()
    time.sleep(10)
```

9.6 Productivity Applications

9.6.1 IoT Printer

This case study is about an IoT printer that prints a daily briefing every morning. The daily briefing comprises of the current weather information, weather predictions for the day and the user's schedule for the day (obtained from the user's Google Calendar account).

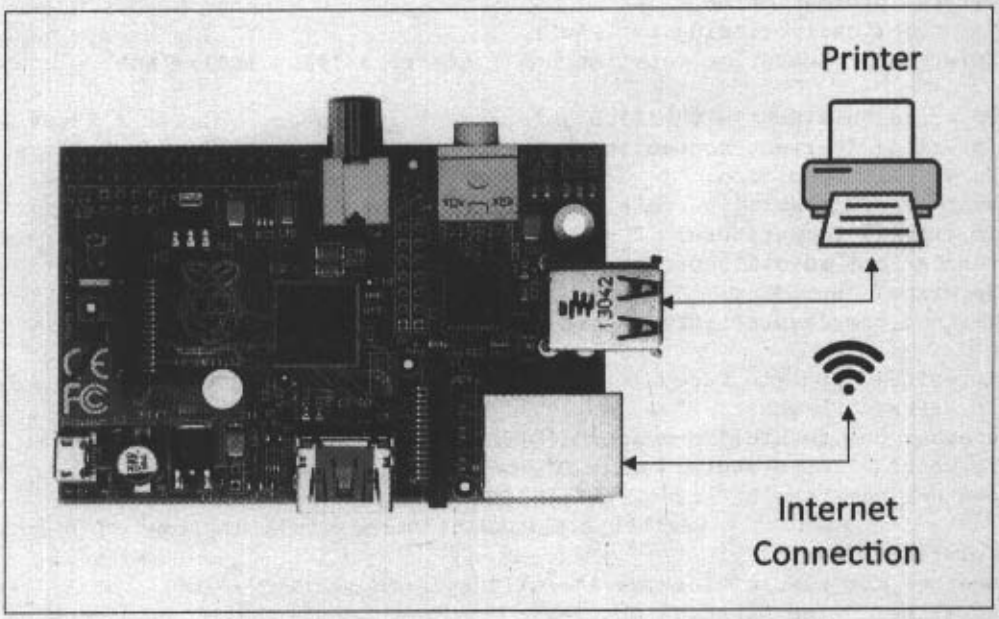


Figure 9.37: Connecting a printer to Raspberry Pi

Box 9.31 shows the code for the service that runs on the mini-computer which is connected to the printer.

Box 9.31: Python code for IoT printer

```
import gflags
import httplib2
import datetime
from apiclient.discovery import build
from oauth2client.file import Storage
from oauth2client.client import OAuth2WebServerFlow
from oauth2client.tools import run
```

```
import pywapi
import popen2


#Get weather information from weather.com
weather_com_result = pywapi.get_weather_from_weather_com('INXX0096')

#Write information to file
fp= file('dailybrieding.txt','w')
fp.write(str(datetime.datetime.now().strftime("%A - %D"))+'\n')

fp.write("WEATHER INFORMATION \n")
fp.write( "Current conditions:\n")
fp.write( "Condition:  "+
weather_com_result['current_conditions']['text']+'\n')
fp.write( "Temperature:  " +
weather_com_result['current_conditions']['temperature']+'\n')
fp.write( "Humidity:  " +
weather_com_result['current_conditions']['humidity']+'\n')

fp.write( "Today's Forecast:\n")
fp.write( "Forecast:  " +
weather_com_result["forecasts"][0]["day"]["brief_text"]+'\n')
fp.write( "Temperature:  Max:  " +
weather_com_result["forecasts"][0]["high"] + ",
          Min:  " + weather_com_result["forecasts"][0]["low"]+'\n')
fp.write( "Humidity:  " +
weather_com_result["forecasts"][0]["day"]["humidity"]+'\n')
fp.write( "Precipitation chances:  " +
weather_com_result["forecasts"][0]["day"]["chance_precip"]+'\n')

   fp.write( "TODAY'S CALENDAR:\n")


#Get calendar information

FLAGS = gflags.FLAGS

# Client_id and client_secret from Google Developers Console
FLOW = OAuth2WebServerFlow(
   client_id='<enter id>',
   client_secret='<enter secret>',
   scope='https://www.googleapis.com/auth/calendar',
   user_agent='MyTestApp/1')
```

```
# Credentials will get written back to a file.
storage = Storage('calendar.dat')
credentials = storage.get()
if credentials is None or credentials.invalid == True:
credentials = run(FLOW, storage)

# Create an httplib2.Http object to handle HTTP requests and authorize
# with good Credentials.
http = httplib2.Http()
http = credentials.authorize(http)

# Build a service object for interacting with the API.
service = build(serviceName='calendar', version='v3', http=http,
    developerKey='<enter key>')

startdatetime =
str(datetime.datetime.now().strftime("%Y-%m-%dT00:00:00+05:30"))
enddatetime =
str(datetime.datetime.now().strftime("%Y-%m-%dT23:59:59+05:30"))

page_token = None
while True:
   events = service.events().list(calendarId='primary',
   pageToken=page_token, timeMin=startdatetime,
   timeMax=enddatetime).execute()
for event in events['items']:
   fp.write("Event:  "+ event['summary'] +
   " From:  "+ event['start']['dateTime'] + " To:  " +
    event['end']['dateTime']+'\n')
   page_token = events.get('nextPageToken')
   if not page_token:
   break

fp.close()

#Print the weather and calendar information file
popen2.popen4("lpr -P Xerox-Phaser-3117 dailybrieding.txt")
```

Box 9.32 shows an example of a daily briefing printed by the IoT printer.

■ Box 9.32: Example of a daily briefing printed by the IoT printer

```
    Friday - 06/20/14

WEATHER INFORMATION

Current conditions:
Condition:  Widespread Dust
Temperature:  42
Humidity:  25

Today's Forecast:
Forecast:
Temperature:  Max:  42, Min:  32
Humidity:  0
Precipitation chances:  0
--------------------

TODAY'S CALENDAR:

Event:  Team Meeting
From:  2014-06-20T15:30:00+05:30 To:  2014-06-20T16:30:00+05:30
```

Summary

In this chapter you learned about various applications of IoT and fully developed case studies. This provided you with a solid foundation, hopefully, that will assist you in designing and implementing various levels of IoT systems. From the smart lighting case study you learned how to implement a level-1 IoT system comprising of a local controller, device and application. Services were implemented using the Django REST framework. From the intrusion detection system case study, you learned about designing the process specification, domain model, information model, service specifications, functional and operational view specifications for a level-2 IoT system. From the weather monitoring system case study you learned about two alternative approaches of implementing the services for an IoT system - one based on REST and other based on WebSocket. The weather monitoring system described is a level-6 IoT system with multiple independent end nodes which perform sensing and send data to the cloud. The REST implementation was done using the Django REST framework and the WebSocket implementation was done using the AutoBahn framework. From the air pollution monitoring, forest fire detection and smart irrigation case studies you learned how to interface various types of sensors with an IoT device and process the sensor data.

Lab Exercises

1. Design and implement a fire alarm IoT system, using a Raspberry Pi device, temperature, CO_2 and CO sensors. Follow the steps below:
   - Define the process specification of the system. The system should collect and analyze the sensor data and send email alerts when a fire is detected.
   - Define a domain model.
   - Define service specifications.
   - Design a deployment of the system. The system can be a level-1 IoT system.
   - Define the functional and operational view specifications.
   - Implement the web services and controller service.
2. For the fire alarm IoT system in exercise-1, identify the configuration and state data. Define a YANG module for the system.
3. Rework the home automation case study to make it a level-2 IoT system.
4. Extend the functionality of the home intrusion detection IoT system to send email alerts when an intrusion is detected.
5. Extend the functionality of the home intrusion detection IoT system by interfacing a webcam. Implement a function in the controller to capture an image from the webcam and send it as an attachment in the email alert when an intrusion is detected.
6. Box 9.25 shown the code for a dummy analytics component of weather monitoring system. Implement the analytics component to compute the hourly maximum and minimum values of temperature and humidity.
7. Implement the air pollution monitoring system using the WebSocket approach.
8. Implement the analytics component for the forest fire detection system.