Chapter 11
Tools for IoT
Outline

• Infrastructure automation & configuration management tools:
  • Chef
  • Puppet

• NETCONF and YANG case studies

• IoT code generator tool
The Purpose of Configuration Management Tools
# Key Differences

<table>
<thead>
<tr>
<th></th>
<th>Chef</th>
<th>Puppet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalability</strong></td>
<td>High Scalability</td>
<td>High Scalability</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>Imperative Language (Ruby)</td>
<td>Declarative Language (JSON/XML)</td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ease of Setup</strong></td>
<td>Not very easy</td>
<td>Not very easy</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Management</strong></td>
<td>Not very easy</td>
<td>Not very easy</td>
</tr>
<tr>
<td><strong>Interoperability</strong></td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td><strong>Terms</strong></td>
<td>Recipes and Cookbooks</td>
<td>Manifests and Modules</td>
</tr>
</tbody>
</table>

**Imperative Language**: programming language that uses statements to change a program’s state.

**Declarative Language**: programming language that expresses the logic of computation without describing the logic flow.
Introduction to Chef

- Chef is an automation tool that converts infrastructure into code

This feature enables Chef to configure and manage multiple systems with ease
Components of Chef

- **Workstation**
  - Consists of **Recipe**
  - Consists of **Knife**

- **Server**
  - Consists of **Cookbook**

- **Node**
  - Consists of **Chef Client**
Terms in Chef

- **Cookbook**: collection of recipes, attributes, templates and resources.
- **Recipe**: configuration element written in Ruby language that specifies various resources to be managed and how to manage the resources.
- **Resource**: fundamental unit of configuration.
- **Knife**: command utility that provides interface between workstation and server.
- **Run Lists**: ordered list of recipes and/or roles.
Architecture and Working of Chef

- **Workstation**
  - knife upload simpli-db

- **Server**

- **Nodes**:
  - Node A
  - Node B
  - Node C
  - Node D
3 steps needed to set up Chef Environment:

1. **Set up Chef Server**
   (Signing up for a free trial Hosted Enterprise Chef and download a starter kit which includes PEM certificates)

2. **Set up Workstation**
   Setting up Chef on workstation by simply run (for Linux workstation):
   ```
curl -L https://www.opscode.com/chef/install.sh | sudo bash
   ```
   Above command will invoke the Chef’s omnibus installer to install all you need to get started with Chef.

3. **Setup Chef-client on nodes**
   Choose in which mode the node will run (cloud-based, physical or virtual). After choosing the mode, the node need to be bootstrapped. This process installs the Chef client and checks in with the Chef server. If Amazon EC2 is used, below is the knife command that need to be run on the Workstation:
   ```
   knife bootstrap <IP-Address> -sudo -x ubuntu -I <keypair.pem> -N <nodeName>
   ```
   For bootstrapping the node, EC2 keypair (PEM) is required.

*Note: There are open source versions that can also be used to setup Chef Environment on your own node.*
There are 3 case studies shown in this Chapter:

1. Multi-tier Architecture Deployment
   Infrastructure that consists of a number of tiers or layers.

2. Hadoop Cluster
   Group of computers that are connected together to store and process large dataset.

3. Storm Cluster
   Real-time big data-processing system.
Multitier Application Architecture consists of
  - Presentation tier is a graphical user interface (GUI)
  - Application tier handles logic
  - Data tier stores information

The most used multitier architecture is the three-tier application
  - The three tiers are logical, not physical, and may run on the same physical server
Three-tier Deployment with Chef

Provision
- Linux Instance
- Linux Instance
- Linux Instance

Configure
- Load Balancer
- App Server
- App Server
- Database Server

Integrate
- Load Balancer
- App Server
- App Server
- Database Server
Hadoop Cluster

• Composed of a network of master and slave nodes
• The master nodes typically utilize higher quality hardware and include
  • NameNode
  • JobTracker.
• The slave nodes consist of virtual machines include
  • DataNode
  • TaskTracker
  Do the actual work of storing and processing the jobs as directed by the master nodes.
Hadoop Cluster with Chef

Provision
- Master Node Linux Instance
- Slave Node Linux Instance
- Slave Node Linux Instance

Configure
- Master Node
  - NameNode
  - JobTracker
- Slave Node
  - DataNode
  - TaskTracker
- Slave Node
  - DataNode
  - TaskTracker
- Slave Node
  - DataNode
  - TaskTracker

Integrate
- Master Node
  - NameNode
  - JobTracker
- Slave Node
  - DataNode
  - TaskTracker
- Slave Node
  - DataNode
  - TaskTracker
- Slave Node
  - DataNode
  - TaskTracker
Master Nodes

- The master nodes include a NameNode and JobTracker
  - NameNode is responsible to:
    - Store the metadata and another data related to DataNode.
    - Managing the file system namespace.
    - Control the access to the data blocks.
    - Check the availability of DataNodes.
  - JobTracker is responsible to:
    - Find the best TaskTracker nodes to execute task based on locality.
    - Monitor TaskTrackers
    - MapReduce execution
The slave nodes include a DataNode and TaskTracker.

DataNode is responsible to:
- Main storage of data
- Storing, creating, deleting jobs according to the instruction of NameNode
- Send health report periodically to NameNode

TaskTracker is responsible to:
- Execute Mapper and Reducer tasks
- Signaling the progress to the JobTracker
Advantages of Hadoop Cluster

- Scalable
- Flexible
- Fast
- Resilient to failure
- Cost effective
Storm Cluster

• Storm is a distributed, open-source and fault-tolerant system for processing streams of data.

• Each component has its own responsibility for a simple specific processing task.
  • The input stream of a Storm cluster is handled by a component called a spout.
  • The spout passes the data to a component called a bolt which then either sends the data in some storage, or passes it to some other bolt.
  • This arrangement of all the components (spout and bolts) and their connections is called a topology.

Architecture of Storm Cluster

Input

Spout

Spout

Bolt

Bolt

Bolt

Bolt

Database

Transform data

Passes data

Passes data

Passes data

Passes data

Data storage

Data storage
Key Properties of Storm

• Extremely broad set of use cases
• Scalable
• Guarantees no data loss
• Extremely robust
• Fault-tolerant

Components of Storm Cluster (1/3)

Nimbus

Zookeeper

Zookeeper

Zookeeper

Supervisor

Supervisor

Supervisor

Supervisor

Components of Storm Cluster (2/3)

• **Nimbus**
  • Manage, coordinate and monitor topologies running on a cluster.
  • Uploads computation for execution
  • Launch workers across the cluster
  • Track the status of all supervisor nodes and the tasks assigned to each.

• **ZooKeeper**
  • ZooKeeper cluster is used by Storm to coordinate various processes.
  • Storage for all of the states associated with the cluster and the various tasks submitted to the Storm.

Components of Storm Cluster (3/3)

• **Supervisor**
  - Supervisor nodes are the worker nodes in a Storm cluster.
  - Each supervisor node runs a supervisor daemon.
  - Supervisor daemon responsible for creating, starting, and stopping worker process.
## Storm Cluster vs Hadoop Cluster

<table>
<thead>
<tr>
<th>Storm Cluster</th>
<th>Hadoop Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time processing</td>
<td>Batch processing</td>
</tr>
<tr>
<td>With ZooKeeper based coordination</td>
<td>Without ZooKeeper based coordination</td>
</tr>
<tr>
<td>Nimbus and Supervisor</td>
<td>JobTracker and TaskTracker</td>
</tr>
<tr>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Shutdown by the user or an unexpected unrecoverable failure</td>
<td>Sequential order system and completed eventually.</td>
</tr>
<tr>
<td>Nimbus dies nothing gets affected</td>
<td>MasterNode dies all the running jobs are lost.</td>
</tr>
</tbody>
</table>
Setting up Storm Cluster (1/8)
Creating cookbook and launch nodes

First step is to create a cookbook named storm with the following command:

```
knife cookbook create storm
```

Name of the cookbook (you can use your own naming)

Launch three Amazon EC2 nodes (Nimbus, ZooKeeper and Supervisor) and bootstrap them with the following command:

```
knife bootstrap <nimbus-IP-address> -sudo -x ubuntu -i mykeypair.pem -N nimbusnode
knife bootstrap <supervisor-IP-address> -sudo -x ubuntu -i mykeypair.pem -N supervisornode
knife bootstrap <zookeeper-IP-address> -sudo -x ubuntu -i mykeypair.pem -N zookeepernode
```

Execute a bootstrap operation with sudo

username

Identity file for authentication

The unique identifier of the node
Recipe for setting up Zookeeper

setup_zk = Mixlib::ShellOut.new("echo \" deb [arch=amd64]
  http://archive.cloudera.com/cdh4/ubuntu/precise/amd64/cdh precise-cdh4 contrib\" >>
  /etc/apt/sources.list.d/cloudera.list", :cwd => '/home/ubuntu')
setup_zk.run_command

setup_zk = Mixlib::ShellOut.new("echo \"deb-src
  http://archive.cloudera.com/cdh4/ubuntu/precise/amd64/cdh precise-cdh4 contrib\" >>
  /etc/apt/sources.list.d/cloudera.list", :cwd => '/home/ubuntu')
setup_zk.run_command

setup_zk = Mixlib::ShellOut.new("apt-get -q -y update", :cwd => '/home/ubuntu')
setup_zk.run_command

setup_zk = Mixlib::ShellOut.new("apt-get install -q -y zookeeper zookeeper-server", :cwd =>
  '/home/ubuntu')
setup_zk.run_command

setup_zk = Mixlib::ShellOut.new("./zkServer.sh start", :cwd => '/usr/lib/zookeeper/bin/‘)
setup_zk.run_command

This recipe is used to generate:
- SSH keys
- Setting up hosts
- Installing Java
- Setting up ZooKeeper
- Setting up Storm
Recipe for setting up Storm (1/3)

require 'chef/shell_out'

download_depend = Mixlib::ShellOut.new("apt-get install build-essential uuid-dev git pkg-config libtool autoconf automake", :cwd => '/home/ubuntu')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("apt-get install unzip", :cwd => '/home/ubuntu')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("wget http://download.zeromq.org/zeromq-2.1.7.tar.gz", :cwd => '/home/ubuntu')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("tar -xzf zeromq-2.1.7.tar.gz", :cwd => '/home/ubuntu')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("./configure", :cwd => '/home/ubuntu/zeromq-2.1.7')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("make", :cwd => '/home/ubuntu/zeromq-2.1.7')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("make install", :cwd => '/home/ubuntu/zeromq-2.1.7')
download_depend.run_command
Recipe for setting up Storm (2/3)

Recipe for setting up Storm (2/3)

download_depend = Mixlib::ShellOut.new("export JAVA_HOME=/usr/lib/jvm/java-7-oracle", :cwd => '/home/ubuntu')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("git clone https://github.com/nathanmarz/jzmq.git", :cwd => '/home/ubuntu')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("touch classdist_noinst.stamp", :cwd => '/home/ubuntu/jzmq/src')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("CLASSPATH=.:./.:$CLASSPATH javac -d . org/zeromq/ZMQ.java org/zeromq/ZMQException.java org/zeromq/ZMQQueue.java org/zeromq/ZMQForwarder.java org/zeromq/ZMQStreamer.java", :cwd => '/home/ubuntu/jzmq/src')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("./autogen.sh", :cwd => '/home/ubuntu/jzmq')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("./configure", :cwd => '/home/ubuntu/jzmq')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("make", :cwd => '/home/ubuntu/jzmq')
download_depend.run_command

download_depend = Mixlib::ShellOut.new("make install", :cwd => '/home/ubuntu/jzmq')
download_depend.run_command

Setting up JAVA_HOME

Clone JZMQ

Create a file

Configure JZMQ

Make and install JZMQ
Recipe for setting up Storm (3/3)

### Download Storm

```ruby
download_storm = Mixlib::ShellOut.new('wget https://dl.dropbox.com/u/133901206/storm-0.8.2.zip', :cwd => '/home/ubuntu')
download_storm.run_command
```

### Unzip the downloaded package

```ruby
download_storm = Mixlib::ShellOut.new('unzip storm-0.8.2.zip', :cwd => '/home/ubuntu')
download_storm.run_command
```

### Create a symbolic links

```ruby
download_storm = Mixlib::ShellOut.new('ln -s storm-0.8.2 storm', :cwd => '/home/ubuntu')
download_storm.run_command
```

### Gives the ownerships to ubuntu

```ruby
download_storm = Mixlib::ShellOut.new('chown -R ubuntu:ubuntu storm', :cwd => '/home/ubuntu')
download_storm.run_command
```

### Setup Directory

```ruby
# setup local directory
directory '/home/ubuntu/stormlocal' do
  owner "ubuntu"
  group "ubuntu"
  action :create
  recursive true
end
```
Chef role for setting up Storm cluster

name "storm_cluster_role"
description "Setup cluster nodes"
run_list [  "recipe[storm::setup_hosts]",  "recipe[storm::ssh_keys]",  "recipe[storm::authorized_nodes]",  "recipe[storm::setup_java]"
]

Chef role for setting up Zookeeper

name "storm_zookeeper_role"
description "Setup zookeeper node"
run_list [  "recipe[storm::setup_zookeeper]"
]

Chef role for setting up nodes for Storm cluster

name "storm_setup_role"
description "Setup storm nodes"
run_list [  "recipe[storm::setup_storm]"
]
The Storm cookbook is then uploaded to the Chef server using the following commands:

```
Knife cookbook upload storm
```

Create roles on the server from the role files using the following commands:

```
Knife role from file storm_cluster_role.rb
Knife role from file storm_zookeeper_role.rb
Knife role from file storm_setup_role.rb
```

Add the roles to the run lists of the Nimbus, ZooKeeper and Supervisor nodes using the following commands:

```
Knife node run_list add nimbusnode 'role[storm_cluster_role]'  
Knife node run_list add nimbusnode 'role[storm_setup_role]'  

Knife node run_list add supervisornode 'role[storm_cluster_role]'  
Knife node run_list add supervisornode 'role[storm_setup_role]'  

Knife node run_list add zookeepernode 'role[storm_cluster_role]'  
Knife node run_list add zookeepernode 'role[storm_zookeeper_role]'  
```
Setting up Storm Cluster (8/8)
Run chef-client from workstation

Chef-client is run on the nimbus, zookeeper and supervisor nodes (from the workstation) as follows:

Knife ssh <nimbus-IP-address> ‘sudo chef-client’ -m -x ubuntu -i mykeypair.pem
Knife ssh <supervisor-IP-address> ‘sudo chef-client’ -m -x ubuntu -i mykeypair.pem
Knife ssh <zookeeper-IP-address> ‘sudo chef-client’ -m -x ubuntu -i mykeypair.pem
Puppet

- Puppet is also a configuration management tool that can be used to manage configurations on a variety of platforms.
- It is usually deployed in a client-server model. Server runs Puppet Master and client runs the Puppet Agents.
- Puppet Master maintains the configuration information for the clients.
- Puppet Agents connect to the master to obtain information on the desired state.
- Uses declarative modeling language.
**Resource Abstraction Layer (RAL):** consist of high-level modules (types) and platform-specific implementations (providers).

**Class:** Define a collection of resources which are managed together as a single unit.

**Manifests:** Puppet programs.

**Module:** Consists of multiple files containing the class definitions.

**Resources:** It is a fundamental unit of configuration. Similar resources are grouped together into resources types.
Multi-tier Deployment with Puppet (1/6)
Directory Structure

• Creating a multi-tier deployment comprising of haproxy, load balancer, Django application server and MongoDB database server will be learned in this case study.

• First of all, create a puppet module with the following directory structure and the files on the Puppet master node:

```
-/etc
  |-puppet
    ||-modules
      |||-threetierdeployment
        ||||-manifests
          ||||||-init.pp
          |||||||-haproxy.pp
          ||||||| |-django.pp
          ||||||| |-mongodb.pp
          ||||||||-templates
          |||||||||-haproxy.cfg.erb
```
Multi-tier Deployment with Puppet (2/6)

haproxy class configuration

On the left is the haproxy class which contains a package resource definition for installing haproxy and file resource for configuration file (haproxy.cfg)

class haproxy{
  $global_options = {
    'chroot' => '/var/lib/haproxy',
    'pidfile' => '/var/run/haproxy.pid',
    'maxconn' => '4000',
    'user' => 'haproxy',
    'group' => 'haproxy',
    'daemon' => '',
    'stats' => 'socket /var/lib/haproxy/stats'
  },
  $defaults_options = {
    'log' => 'global',
    'stats' => 'enable',
    'option' => 'redispatch',
    'retries' => '3',
    'timeout' => [
      'http-request 10s',
      'queue 1m',
      'connect 10s',
      'client 1m',
      'server 1m',
      'check 10s',
    ],
    'maxconn' => '8000'
  },
}

{ package { 'haproxy':
    ensure => present,
  }

  file { 'haproxy-config':
    path => '/etc/haproxy/haproxy.cfg',
    content => template('/etc/puppet/modules/threetierdeployment/haproxy.cfg.erb'),
  }
}
Multi-tier Deployment with Puppet (3/6)
Puppet Configuration File

On the left is the template for the configuration file.

```erb
Global
  <% @global_options.each do |key,val| -%>
  <% if val.is_a?(Array) -%>
    <% val.each do |item| -%>
      <%= key %>  <%= item %>
    <% end -%>
  <% else -%>
    <%= key %>  <%= val %>
  <% end -%>
<% end -%>

defaults
  <% @defaults_options.each do |key,val| -%>
  <% if val.is_a?(Array) -%>
    <% val.each do |item| -%>
      <%= key %>  <%= item %>
    <% end -%>
  <% else -%>
    <%= key %>  <%= val %>
  <% end -%>
<% end -%>
```
Below is the Django class that contains package resources for python-pip provider and Django.

class django{
    package { 'python-pip':
        ensure => installed,
    }
    package { 'django':
        ensure => installed,
        provider => 'pip',
        require => Package["python-pip"],
    }
}

Multi-tier Deployment with Puppet (4/6)
Django class configuration
Below is the Mongodb class that commands for setting up MongoDB.

class mongodb{
  exec { "cmd1":
    command => "/usr/bin/apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10",
  }

  exec { "cmd2":
    command => "/bin/echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' $>>$ /etc/apt/sources.list.d/10gen.list",
    require => Exec["cmd1"],
  }

  exec { "cmd3":
    command => "/usr/bin/apt-get update",
    require => Exec["cmd2"],
  }

  exec { "cmd4":
    command => "/usr/bin/apt-get install mongodb-10gen",
    require => Exec["cmd3"],
  }
}
To apply the Puppet module on the clients nodes, run the Puppet agent on each client node as follows:

```
sudo puppet agent -t
```
IoT System Management with NETCONF-YANG

- Management System
- Management API
- Transaction Manager
- Rollback Manager
- Data Model Manager
- Configuration Validator
- Configuration Database
- Configuration API
- Data Provider API
Steps for IoT device Management with NETCONF-YANG

1. Create a YANG model of the system
2. Build the callbacks
3. Load the YANG module
4. Compile the YANG model with ‘Inctool’
5. Connect from the management system to the Netopeer server
6. Fill in the IoT device management code in the Trans API module
7. NETCONF commands can be issued
NETCONF-YANG Case Study

• Managing Smart Irrigation IoT System
• Managing Home Intrusion Detection IoT System
The flow behind this case study:
- This case study uses an IoT device and soil moisture sensors.
- The sensor is used to measure the amount of moisture in the soil.
- If the level of moist above threshold, it will release the flow of water through the irrigation pipes.

Callbacks on soil moisture sensor, IoT device and action
IoT device configuration and soil moisture sensors threshold configuration
Compile the YANG model with 'Inctool'
Irrigation system code in the Trans API C module

Load the module
Connect the Irrigation management system to the Netopeer server
NETCONF commands can be issued
Home Intrusion Detection IoT System

- The purpose is to detect intrusion using sensors.
  - PIR sensor and door sensor.
  - Will raise alerts if intrusion detected.

PIR and door sensor threshold configuration → Compile the YANG model with ‘Inctool’

Code to control Home Intrusion system included in Trans API C module

Callbacks to call the sensors and action → Load the module → Connect Home Intrusion management system to the Netopeer server

NETCONF commands can be issued
IoT Code Generator Levels (1/2)

**Level-1**

- **Local**
  - App
  - REST Services
  - Database
  - Controller Service
  - Resource
  - Device

- **Cloud**
  - Monitoring Node performs analysis, stores data

In this level there is a single node that stores data, performs analysis and hosts the app.

**Level-2**

- **Local**
  - App
  - Controller Service
  - Resource
  - Device
  - Database

- **Cloud**
  - REST Calls
  - Cloud Storage

In this level there is a single node that performs local analysis. Data is stored in the cloud and app is cloud-based.

**Level-3**

- **Local**
  - App
  - Controller Service
  - Resource
  - Device

- **Cloud**
  - REST Calls
  - Cloud Storage & Analysis

In this level there is a single node. Data is stored and analyzed in the cloud and app is cloud-based.
IoT Code Generator Levels (2/2)

**Level-4**

- **Local**
  - Controller Service
  - Resource
  - Device
  - Monitoring Nodes perform local analysis

- **Cloud**
  - App
  - REST Services
  - Database
  - Cloud Storage

**Level-5**

- **Local**
  - Controller Service
  - Resource
  - Endpoint Device
  - Coordinator Device

- **Cloud**
  - App
  - REST Services
  - Database
  - Coordinator
  - Routers/End Points
  - Cloud Storage & Analysis

In this level there are multiple nodes that perform local analysis. Data is stored in the cloud and app is cloud-based.

In this level there are multiple nodes. Data is stored and analyzed in the cloud and app is cloud-based.
To begin with, the user selects an IoT level for the system for which the code is to be generated. The next step is to select an IoT device as shown below.
Select the sensors that will be used.
Storage option is selected as shown below.
Storage option is selected and configured as shown below.
The code generator generates the controller code and app code.
import time
import datetime
import MySQLdb

#Initialize MySQL
con = MySQLdb.connect(host = 'localhost', user = 'iot', passwd = 'iot', db = 'iot', port = 3306)
cur = con.cursor()
cur.execute("CREATE TABLE data(id INTEGER PRIMARY KEY AUTOINCREMENT, Timestamp VARCHAR(50) NOT NULL, SensorName VARCHAR(50) NOT NULL, Data VARCHAR(100) NOT NULL)")

#Controller setup function
def setupController():
    return true

#Controller main function
def runController():
    timestamp=datetime.datetime.utcnow()
    print timestamp
    return true

setupController() while True:
    runController()
    time.sleep(10)
The code generator generates the controller code and app code.

```python
from django.shortcuts import render_to_response
from django.template import RequestContext
import requests
import json

def home(request):
    # Get dht22 data (for id=1)
    r = requests.get('http://service-endpoint/dht22/1', auth=('username', 'password'))
    result = r.text
    output = json.loads(result)
    dht22timestamp=output['Timestamp']
    dht22data=output['Data']
```

Download views.py
def home(request):
    return render_to_response('template.html',{}, context_instance=RequestContext(request))

urlpatterns = patterns(,
    url(r'^home/', 'myapp.views.home'), 
)

<!DOCTYPE HTML>
<html>
<head>
    <title>App</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>
<h2>Sensor Data</h2>
<table width="50%">
    <tr>
        <td>Sensor Name</td>
        <td>Timestamp</td>
        <td>Data</td>
    </tr>
</table>
</body>
</html>
Below are the wizards for generating the services code.
Below are the wizards for generating the services code.
The generated service code shown below.

```python
from django.db import models

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

from rest_framework import serializers
```
Advantages:
• It provides a simple code skeleton
• Do not need to build the code from scratch

Current Disadvantages:
• Only Raspberry Pi is supported
• Limited types of sensors
• Limited types of local and cloud data storage
Summary

• Chef
  • Multi-tier Deployment
  • Hadoop Cluster
  • Storm Cluster
• Puppet
  • Multi-tier Deployment

• NETCONF-YANG
  • Managing Smart Irrigation IoT System
  • Managing Home Intrusion Detection IoT System
• IoT Code Generator