# Introduction to Pyjamas, Part 1: Exploit the synergy of GWT and Python

Rick Hightower (richardhightower@gmail.com)
Developer
Mammatus Inc.

09 March 2010

Pyjamas is a cool tool, or framework, for developing Asynchronous JavaScript and XML (Ajax) applications in Python. It's a versatile tool that you can use to write comprehensive applications without writing any JavaScript code. This series examines the myriad aspects of Pyjamas, and this first article explores Pyjamas's background and basic elements. Step through the process of building a sample application with Pyjamas using the article's examples.

View more content in this series

## Introduction

Google's Web Toolkit (GWT) lets you develop a Rich Internet Application (RIA) with Ajax, entirely in Java™ code. You can use the rich Java toolset (IDEs, refactoring, code completion, debuggers, and so on) to develop applications that can be deployed on all major Web browsers. With GWT you can write applications that behave like desktop applications but run in the browser. Pyjamas, a GWT port, is a tool and framework for developing Ajax applications in Python.

Pyjamas contains a stand-alone Python-to-JavaScript compiler, and an Ajax framework and widget set. Using these components, you can write comprehensive applications without writing a single line of JavaScript.

This article explains the background, rationale, related tools, and benefits of Pyjamas while showing you how to create a sample application that will store basic contact information (name, e-mail address, telephone number). You can also download the code for the sample application.

Part two in this series will explain how to build custom Pyjamas components.

## Background

Python was one of the first popular languages ported to the JVM (Jython) and later to .Net (IronPython). Python-like syntax has been ported to produce machine code that's comparable to writing programs in C (Cython). It's no surprise, then, that Python is one of the first languages

(after Google blazed the trail with the Java language) to be translated to JavaScript to run in a cross-browser manner.

### Mighty XUL

In 2009, Pyjamas-Desktop (now part of Pyjamas proper) was also ported to work with XUL. XUL is to Firefox as WebKit is to Safari, more or less. You can run Pyjamas on top of XUL. Reportedly, it took only two days to port Pyjamas to XUL thanks to project Hulahop (from the OLPC Sugar team) and to the developers of python-xpcom.

In the recent past, the chances of doing an entire application in Ajax seemed remote. With GWT, however, you can develop an RIA with Ajax entirely in Java code. GWT lets you write applications that behave like desktop applications but run in the browser.

Conversely, Adobe AIR and Silverlight allow Web-style applications to run on the desktop. Android, Adobe AIR, Google Chrome, Safari, and iPhone all use WebKit for rendering. One problem with GWT is that it does not allow you to write applications that run as desktop applications (even though GWT's development toolset for rendering is based on WebKit).

Pyjamas has a Python-to-JavaScript compiler similar to GWT's, and a set of Ajax widgets that have an identical API to their GWT counterparts. (You can actually use the GWT documentation to develop Pyjamas applications.) Python has a terse, powerful syntax; for example, GWT 1.2 took 80,000 lines of code to write, yet Pyjamas took a mere 8,000 to do the same task.

## Overview of Pyjamas

### Problems with XUL and WebKit Python bindings

The MSHTML port is seemingly the best of the lot, and the underlying Python bindings to WebKit and XUL are in flux. When the WebKit team does not port the Python bindings to WebKit GTK, it causes untold pain and suffering.

At times it appears that WebKit and the xulrunner Python bindings are being undermined or at least neglected.

Remember that Pyjamas-Desktop is not tied exclusively to WebKit. Pyjamas brings WebKit, XUL, and MSHTML to Python developers. Thus, Pyjamas-Desktop can use any one of the three browser engines. Through them, Pyjamas becomes both a cross-browser and a cross-platform GUI widget set.

WebKit, XUL, and their ilk bring modern flair to desktop applications. Pyjamas brings WebKit to Python developers. With Webkit, Pyjamas becomes a cross-browser and cross-platform set of GUI widgets. You can develop widgets that will run anywhere WebKit and XUL run. The Pyjamas API-based application can live anywhere GWT applications would live. Plus, Pyjamas lets you write desktop applications built on top of WebKit and XUL. This is preferable to building applications on top of Qt or GTK because WebKit supports CSS, and it is used in many other places for reliable rendering (iPhone, Safari, Android, and so on). However, there is a bit of trouble with XUL and WebKit regarding Python (see the sidebar).

Pyjamas, like GWT, is a GUI component framework. If you have worked with Swing or GWT, Pyjamas development should feel familiar. Like most GUI frameworks, Pyjamas is event driven.

With Pyjamas you create containers, then add widgets to the containers. The widgets can be labels, text fields, buttons, and so forth. Widgets, like buttons, have event handlers so you can listen for click events from the button.

It's easy to develop with Pyjamas because you can use the same debugging tools that you usually use for Python. Examples include unit tests, print statements, and Python debugger (pdb, a command-line debugger). You can even use Eclipse's Python support for debugging. Keep in mind that you can write Pyjamas applications that run as native Python applications. You do not have to translate Pyjamas applications to JavaScript. You can use Pyjamas just like any other Python GUI toolkit.

The first version of the GUI for the sample application in this article was developed using just Python running from the command line. It was not even initially deployed to the Web, but ran as a desktop application. This is a great advantage for developing RIA applications, because being able to easily debug your programs is a huge benefit.

When you're ready to deploy your application to the Web, you need to be a bit more careful about what library you include. It is common to use JavaScript Object Notation (JSON)-RPC services from a Pyjamas application running in the browser.

## Prerequisites

To build the sample application in this article, you need to download and install Pyjamas. This is not a minor task. After trying to get Pyjamas running on Ubuntu, and failing miserably, I gave up and installed it on Debian. (Rumor has it that Pyjamas also runs on Windows® quite well.) The installed version worked perfectly on Debian. The installation process is likely to be in flux for a while, so you should follow the latest instructions for your environment from the Pyjamas site (see Resources).

To build the services tier, MySQL, Apache, mod_python, and Python JSON-RPC were used.

## Building a sample application

The sample contact-management application stores basic contact information, such as a name, e-mail address, and telephone number. You'll start with a simple Create, Read, Update, and Delete (CRUD) application and later add real storage. You can do the whole thing in one simple Python script with an in-memory "database." The sample uses a service layer, and later replaces this in-memory service layer version with a JSON-backed service layer version that uses MySQL to store the contact information in a relational database.

### Divide and conquer

I prefer to develop a complete GUI talking to a mock layer to separate the GUI development from persistence and from the business logic layer. That way, I can focus on the GUI logic without worrying about debugging remote RPC and so on.

To understand how the mock service is written, you must understand how the runtime application will run. The JSON service will be called asynchronously. When you compile the Pyjamas application into an RIA application (HTML and JavaScript code), when it makes calls the Ajax call

will return the results asynchronously. Therefore, when you build the mock service you'll mock the Ajax lib calling your GUI back asynchronously. Listing 1 illustrates that the `ContactService` calls `callback` methods to the GUI, which is shown later. This is to simulate the JSON asynchronous behavior, which will be added later.

## Listing 1. Contact Service

```
class Contact:
    def __init__(self, name="", email="", phone=""):
        self.name = name
        self.email = email
        self.phone = phone

class ContactService:
    def __init__(self, callback):
        self.callback = callback
        self.contacts = []

    def addContact(self, contact):
        self.contacts.append(contact)
        self.callback.service_eventAddContactSuccessful()

    def updateContact(self, contact):
        self.callback.service_eventUpdateContactSuccessful()

    def removeContact(self, contact):
        self.contacts.remove(contact)
        self.callback.service_eventRemoveContactSuccessful()

    def listContacts(self):
        self.callback.service_eventListRetrievedFromService(self.contacts)
```

The `Contact` class simply represents a contact (name, e-mail, phone number). The `ContactService` has only an in-memory list (not persisting to disk) of contacts. This simple class lets you develop the GUI and later, with slight modification, test the GUI with the real JSON service after you develop the display logic.

The `ContactService` notifies the `ContactListGUI` (defined in Listing 2) of service events using methods that begin in `service_eventXXX`.

`ContactListGUI`, which is only 125 lines long and fairly straightforward, manages nine GUI widgets. It also collaborates with `ContactService` to manage a CRUD listing, as shown in Listing 2.

## Listing 2. ContactListGUI

```
import pyjd # this get stripped out for JavaScript translation
from pyjamas.ui.RootPanel import RootPanel
from pyjamas.ui.Button import Button
from pyjamas.ui.Label import Label
from pyjamas import Window

from  pyjamas.ui.Grid import Grid
from  pyjamas.ui.Hyperlink import Hyperlink
from  pyjamas.ui.TextBox import TextBox

# Constants
CONTACT_LISTING_ROOT_PANEL = "contactListing"
CONTACT_FORM_ROOT_PANEL = "contactForm"
CONTACT_STATUS_ROOT_PANEL = "contactStatus"
CONTACT_TOOL_BAR_ROOT_PANEL = "contactToolBar"
```

```
EDIT_LINK = 3
REMOVE_LINK = 4

#Service code removed

class ContactListGUI:

    def __init__(self):
        self.contactService = ContactService(self)
        self.currentContact = Contact("Rick", "rhightower@gmail.com", "555-555-5555")
        self.addButton = Button("Add contact", self.gui_eventAddButtonClicked)
        self.addNewButton = Button("Add new contact", self.gui_eventAddNewButtonClicked)
        self.updateButton = Button("Update contact", self.gui_eventUpdateButtonClicked)

        self.nameField = TextBox()
        self.emailField = TextBox()
        self.phoneField = TextBox()
        self.status = Label()
        self.contactGrid = Grid(2,5)
        self.contactGrid.addTableListener(self)

        self.buildForm()
        self.placeWidgets()
        self.contactService.listContacts()


    def onCellClicked(self, sender, row, cell):
        print "sender=%s row=%s cell=%s" % (sender, row, cell)
        self.gui_eventContactGridClicked(row, cell)

    def onClick(self, sender):
        if sender == self.addButton:
            self.gui_eventAddButtonClicked()
        elif sender == self.addNewButton:
            self.gui_eventAddNewButtonClicked()
        elif sender == self.updateButton:
            self.gui_eventUpdateButtonClicked()

    def buildForm(self):
        formGrid = Grid(4,3)
        formGrid.setVisible(False)

        formGrid.setWidget(0, 0, Label("Name"))
        formGrid.setWidget(0, 1, self.nameField);

        formGrid.setWidget(1, 0, Label("email"))
        formGrid.setWidget(1, 1, self.emailField)

        formGrid.setWidget(2, 0, Label("phone"))
        formGrid.setWidget(2, 1, self.phoneField)

        formGrid.setWidget(3, 0, self.updateButton)
        formGrid.setWidget(3, 1, self.addButton)

        self.formGrid = formGrid

    def placeWidgets(self):
        RootPanel(CONTACT_LISTING_ROOT_PANEL).add(self.contactGrid)
        RootPanel(CONTACT_FORM_ROOT_PANEL).add(self.formGrid)
        RootPanel(CONTACT_STATUS_ROOT_PANEL).add(self.status)
        RootPanel(CONTACT_TOOL_BAR_ROOT_PANEL).add(self.addNewButton)

    def loadForm(self, contact):
        self.formGrid.setVisible(True)
        self.currentContact = contact
        self.emailField.setText(contact.email)
        self.phoneField.setText(contact.phone)
```

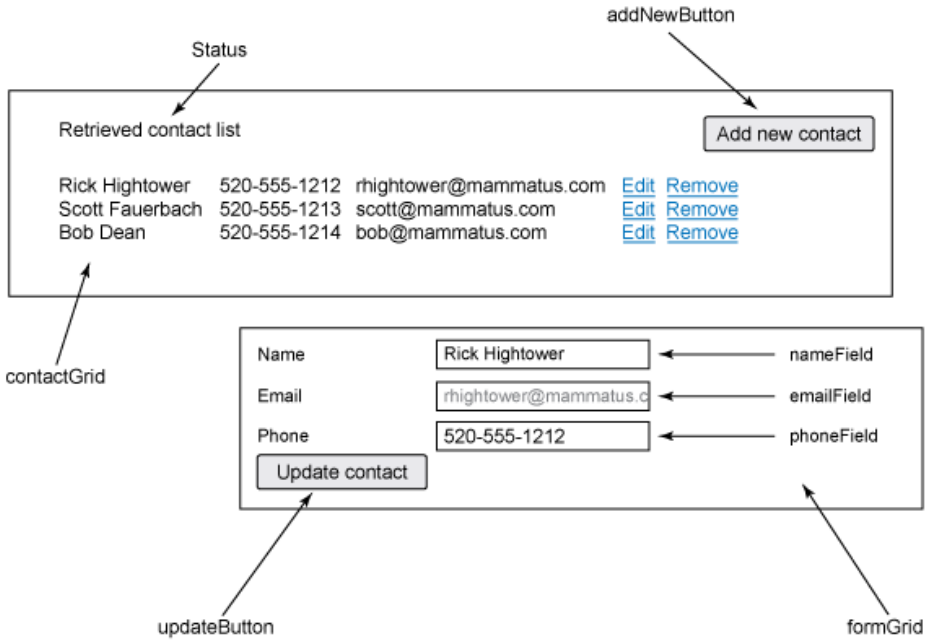
```
        self.nameField.setText(contact.name)

    def copyFieldDateToContact(self):
        self.currentContact.email = self.emailField.getText()
        self.currentContact.name = self.nameField.getText()
        self.currentContact.phone = self.phoneField.getText()
```

The `ContactListGUI init` method calls the `buildForm` method to create a new form grid and populate it with fields to edit contact data. The `init` method then calls the `placeWidgets` method, which places the `contactGrid`, `formGrid`, `status`, and `addNewButton` widgets into slots defined in the HTML page that hosts this GUI application. This is defined in Listing 3.

Figure 1 shows an overview of the widgets being used in the contact management application.

## Figure 1. Widgets in the contact management GUI




## Listing 3. ContactListGUI GUI event handlers

```
<html>
    <head>
      <meta name="pygwt:module" content="Contacts">
      <link rel='stylesheet' href='Contacts.css'>
      <title>Contacts</title>
    </head>
    <body bgcolor="white">

      <script language="javascript" src="bootstrap.js"></script>

      <h1>Contact List Example</h1>

      <table align="center">
      <tr>
        <td id="contactStatus"></td>
      </tr>
      <tr>
```

```
        <td id="contactToolBar"></td>
      </tr>
      <tr>
        <td id="contactForm"></td>
      </tr>
      <tr>
        <td id="contactListing"></td>
      </tr>
      </table>
    </body>
</html>
```

The constants (such as `CONTACT_LISTING_ROOT_PANEL="contactListing"`) correspond to IDs of elements (such as `id="contactListing"`) defined in the HTML page. This allows a page designer to have more control over the layout of application widgets.

The basic application is now built. The next section walks through a couple of common usage scenarios.

## Showing a listing on page load

When the example application's page first loads, it calls the `ContactListEntryPoint`'s `__init__` method. The `__init__` method calls the `ContactServiceDelegate`'s `listContacts` method, which calls the service's `listContact` method asynchronously. The mock `ContactService`'s `listContact` method calls the service-event handler method called `service_eventListRetrievedFromService`, as shown in Listing 4.

### Listing 4. ContactListGUI: Calling the listContact event handler

```
class ContactListGUI:
    …
    def service_eventListRetrievedFromService(self, results):
        self.status.setText("Retrieved contact list")
        self.contacts = results;
        self.contactGrid.clear();
        self.contactGrid.resizeRows(len(self.contacts))
        row = 0

        for contact in results:
            self.contactGrid.setWidget(row, 0, Label(contact.name))
            self.contactGrid.setWidget(row, 1, Label (contact.phone))
            self.contactGrid.setWidget(row, 2, Label (contact.email))
            self.contactGrid.setWidget(row, EDIT_LINK, Hyperlink("Edit", None))
            self.contactGrid.setWidget(row, REMOVE_LINK, Hyperlink("Remove", None))
            row += 1
```

The `service_eventListRetrievedFromService` event-handler method stores the contact list sent by the server. It then:

- Clears the `contactGrid` that displays the contact listing.
- Resizes the number of rows to match the size of the contact list returned from the server.
- Iterates through the contact list, placing name, telephone, and e-mail data for each contact into the first three columns of each row.
- Provides an Edit link and a Remove link for each contact, enabling users to easily remove and edit contacts.

## Editing an existing contact

When a user clicks on an Edit link from the contacts listing, the `gui_eventContactGridClicked` is called, as shown in Listing 5.

## Listing 5. ContactListGUI's gui_eventContactGridClicked event handler method

```
class ContactListGUI:

    …
    def gui_eventContactGridClicked(self, row, col):
         contact = self.contacts[row]
         self.status.setText("Name was " + contact.name + " clicked ")
         if col==EDIT_LINK:
             self.addNewButton.setVisible(False)
             self.updateButton.setVisible(True)
             self.addButton.setVisible(False)
             self.loadForm(contact)
         elif (col==REMOVE_LINK):
             self.contactService.removeContact(contact)

    …
    def loadForm(self, contact):
        self.formGrid.setVisible(True)
        self.currentContact = contact
        self.emailField.setText(contact.email)
        self.phoneField.setText(contact.phone)
        self.nameField.setText(contact.name)
```

The `gui_eventContactGridClicked` method determines whether the Edit link or the Remove link has been clicked by finding out which column was clicked. It then hides `addNewButton` and `addButton`, and makes `updateButton` visible. The `updateButton` displays in the `formGrid` and lets the user send the update information back to `ContactService`. `gui_eventContactGridClicked` then calls `loadForm` (shown in Listing 5), which:

- Sets the `formGrid` to visible.
- Sets the contact being edited.
- Copies the contact properties into `emailField`, `phoneField`, and `nameField` widgets.

When the user clicks the Update button, the `gui_eventUpdateButtonClicked` event-handler method is called, as shown in Listing 6. This method:

- Makes the `addNewButton` visible, so the user can add new contacts.
- Hides the `formGrid`.
- Calls `copyFieldDateToContact`, which copies the text from the `emailField`, `phoneField`, and `nameField` widgets back into the properties of the `currentContact`.
- Calls the ContactServiceDelegate `updateContact` method to pass the newly updated contact back to the service.

## Listing 6. ContactListGUI's gui_eventUpdateButtonClicked event-handler method

```
class ContactListGUI:

    …

    def gui_eventUpdateButtonClicked(self, sender):
        self.addNewButton.setVisible(True)
        self.formGrid.setVisible(False)
        self.copyFieldDateToContact()
        self.contactService.updateContact(self.currentContact)

    def copyFieldDateToContact(self):
        self.currentContact.email = self.emailField.getText()
        self.currentContact.name = self.nameField.getText()
        self.currentContact.phone = self.phoneField.getText()
```

The two scenarios above illustrate how the application works, and how it draws upon the infrastructure provided by App Engine for Java. The rest of the GUI event handlers for `ContactListGUI` are shown in Listing 7, and the rest of the service callback handlers are shown in Listing 8.

## Listing 7. ContactListGUI's gui_eventUpdateButtonClicked event-handler method

```
class ContactListGUI:

    …
    def gui_eventContactGridClicked(self, row, col):
         contact = self.contacts[row]
         self.status.setText("Name was " + contact.name + " clicked ")
         if col==EDIT_LINK:
             self.addNewButton.setVisible(False)
             self.updateButton.setVisible(True)
             self.addButton.setVisible(False)
             self.loadForm(contact)
         elif (col==REMOVE_LINK):
             self.contactService.removeContact(contact)


    def gui_eventAddButtonClicked(self, sender):
        self.addNewButton.setVisible(True)
        self.formGrid.setVisible(False)
        self.copyFieldDateToContact()
        self.contactService.addContact(self.currentContact)

    def gui_eventUpdateButtonClicked(self, sender):
        self.addNewButton.setVisible(True)
        self.formGrid.setVisible(False)
        self.copyFieldDateToContact()
        self.contactService.updateContact(self.currentContact)


    def gui_eventAddNewButtonClicked(self, sender):
        self.addNewButton.setVisible(False)
        self.updateButton.setVisible(False)
        self.addButton.setVisible(True)
        self.loadForm(Contact())
```

## Listing 8. ContactListGUI service call back methods

```
class ContactListGUI:
```

```
    …
    def service_eventListRetrievedFromService(self, results):
        self.status.setText("Retrieved contact list")
        self.contacts = results;
        self.contactGrid.clear();
        self.contactGrid.resizeRows(len(self.contacts))
        row = 0

        for contact in results:
            self.contactGrid.setWidget(row, 0, Label(contact.name))
            self.contactGrid.setWidget(row, 1, Label (contact.phone))
            self.contactGrid.setWidget(row, 2, Label (contact.email))
            self.contactGrid.setWidget(row, EDIT_LINK, Hyperlink("Edit", None))
            self.contactGrid.setWidget(row, REMOVE_LINK, Hyperlink("Remove", None))
            row += 1

    def service_eventAddContactSuccessful(self):
        self.status.setText("Contact was successfully added")
        self.contactService.listContacts()

    def service_eventUpdateContactSuccessful(self):
        self.status.setText("Contact was successfully updated")
        self.contactService.listContacts()

    def service_eventRemoveContactSuccessful(self):
        self.status.setText("Contact was removed")
        self.contactService.listContacts()
```

## Compiling the example

You can compile this example application and run it natively in any modern browser. However, trying to debug an RIA application running in a browser is not fun. Luckily, you can run the whole application as a native Python application by using the Pyjamas-Desktop, as shown in Listing 9.

## Listing 9. Running Pyjamas-Desktop

```
import pyjd # this get stripped out for JavaScript translation
...
if __name__ == '__main__':
    pyjd.setup("public/Contacts.html")
    contacts = ContactListGUI()
pyjd.run()
```

The code in Listing 9 instantiates a Python desktop application and then starts the desktop by calling the `run` method. When you run this application as a desktop application, you can debug it using pdb or Python IDEs that support visual debugging.

I installed Pyjamas in a tools directory under the home directory. When you use the Python debugger, be sure you add the Pyjamas and Pyjamas-Desktop libs to your path, as shown in listing 10.

## Listing 10. Adding Pyjamas to the PYTHONPATH

```
export PYTHONPATH=/home/rick/tools/pyjamas:/home/rick/tools/pyjamas/library
```

When you're done writing your application, you can then run `pyjsbuild` to compile the application into HTML, JavaScript, and JSON-RPC. Listing 11 shows an example script to run the `pyjsbuild`.

### Listing 11. build.sh

```
#!/bin/sh

options="$*"
#if [ -z $options ] ; then options="-O";fi
~/tools/pyjamas/bin/pyjsbuild --print-statements $options Contacts.py
```

When the application is compiled, all you have to do is host the /output folder by a Web server. The example uses a fresh installation of Debian, so apache2 and mod_python were installed with `apt-get`, as shown in Listing 12.

### Listing 12. Installing apache2 and mod_python

```
$sudo apt-get install apache2 libapache2-mod-python
```

mod_python will be used in the next version of the contact listing. The example application was created under /home/rick/tools/pyjamas/examples/contact1. To host this with Apache, add the following code to the Apache httpd.conf file (on Debian this is installed under /etc/apache2).

### Listing 13. /etc/apache2/httpd.conf

```
Alias /pj "/home/rick/tools/pyjamas"
<Directory "/home/rick/tools/pyjamas">
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order deny,allow
    allow from all
</Directory>
```

## Adding JSON-RPC support

### Tips about mod_python and sqllite3

Initially, it seemed there'd be a small service for the small example in this article, and it wouldn't need a unit test or logging. Big mistake! I first tried to use sqllite3 (because it ships with Python) and had some locking issues, which prompted the switch to MySQL. Some lessons about mod_python, JSON-RPC, and sqllite3:

- sqlite3 locks files in strange ways when run locally with unit tests and then later as `su` under apache.
- It's hard to debug in mod_python because you don't get error messaging. Logging and unit tests are essential.

I switched to MySQL, used logging with robust exception handling, and wrote unit tests. The example likely could not have been finished without unit tests and logging. You could probably use sqlite3 if you always run the unit tests with sudo. Or, you might devise other ways to avoid the locking issues.

After you get the GUI logic working, it's time to start programming the JSON-RPC service, which is implemented in Python. JSON-RPC is a standard; you can use any programming language to implement the server side. This way, a Pyjamas front-end app can be fitted into existing projects that have a JSON-RPC back-end Web service. JSON is a data-interchange format. It uses two structures:

- A collection of name/value pairs (a dictionary in Python, a hash table in Java code, or an associative array for Perl)
- Arrays

JSON-RPC is a remote procedure call protocol that uses JSON to encode and marshal arguments and return types. The JSON-RPC project has bindings for Python. Twisted, Django, and many other Python frameworks also have support for JSON-RPC. An easy way to get JSON-RPC is shown in Listing 14.

## Listing 14. Installing JSON-RPC

```
$ svn checkout
   http://svn.json-rpc.org/trunk/python-jsonrpc

$ cd python-jsonrpc
$ python setup.py install
```

To write a JSON-RPC service you annotate method calls with `@ServiceMethod` and then expose a module variable called service that points to the instance that you want to expose using JSON-RPC. Listing 15 shows an example.

## Listing 15. ContactService: JSON-RPC service for the contacts list

```
import logging

logging.basicConfig(filename="/tmp/contactjson.log",
                    level=logging.DEBUG)


logging.debug("Loading contact service")

from jsonrpc import ServiceMethod

use_mysql=True

if use_mysql:
    import MySQLdb as db_api
    logging.debug("Using mysql")
else:
    import sqlite3 as db_api
    logging.debug("Using sqllite3")


db_url = "/tmp/contacts"


class ContactService:

    @ServiceMethod
    def test(self):
        logging.info("Test called")
        return "test"

    def connection(self):
        if use_mysql:
            connection =  db_api.connect(passwd="mypassword", db="contactdb", user="root")
        else:
            connection =  db_api.connect(db_url)
        return connection

    def run_update(self, func):
```

```
        connection = self.connection()
        cursor = connection.cursor()
        try:
            func(cursor)
            cursor.close()
            connection.commit()
        except Exception, e:
            connection.rollback()
            logging.exception("problem handling update")
            raise e
        finally:
            connection.close()

    def run_query(self, func):
        connection = self.connection()
        cursor = connection.cursor()
        lst = None
        try:
            func(cursor)
            lst = cursor.fetchall()
            cursor.close()
        except Exception, e:
            logging.exception("problem handling query")
            raise e
        finally:
            connection.close()
        return lst

    @ServiceMethod
    def addContact(self, contact):
        logging.debug("Add contact called %s", `contact`)
        def ac(cursor):
            if use_mysql:
                cursor.execute("""
                  insert into contact
                          (name, phone, email)
                  values (%(name)s, %(phone)s, %(email)s)
                  """, contact)
            else:
                cursor.execute("""
                  insert into contact
                          (id, name, phone, email)
                  values (NULL, :name, :phone, :email)
                  """, contact)
        self.run_update(ac)


    @ServiceMethod
    def updateContact(self, contact):
       logging.debug("Update contact called %s", `contact`)
       def uc(cursor):
           if use_mysql:
               cursor.execute("""
                  update contact
                          set name = %(name)s, email = %(email)s, phone = %(phone)s
                  where id=%(id)s;
                  """, contact)
           else:
               cursor.execute("""
                  update contact
                          set name = :name, email = :email, phone = :phone
                  where id=:id;
                  """, contact)

       self.run_update(uc)
```

```
    @ServiceMethod
    def removeContact(self, contact):
       logging.debug("Remove contact called %s", `contact`)
       def uc(cursor):
           if use_mysql:
               cursor.execute("delete from contact where id=%(id)s;", contact)
           else:
               cursor.execute("delete from contact where id=:id;", contact)
       self.run_update(uc)


    @ServiceMethod
    def listContacts(self):
        logging.debug("list contact called")
        def lc(cursor):
            cursor.execute("select name, phone, email, id from contact")
        lst = self.run_query(lc)
        def toMap(x):
            return {"name":x[0],"phone": x[1], "email":x[2], "id":x[3]}
        return map(toMap, lst)


service = ContactService()

#If you can't get mod_python working
# you can use CGI with the following line.
#handleCGI(service)
# You have to import handleCGI from jsonrpc
```

Listing 15 can use either MySQL, which is easy to install, or sqlite3, which ships with Python. To use sqlite3 set `use_mysql` to False.

Listing 16 shows a unit test to test this service, which was essential to developing the example application. The listing shows a utility class that gets used by the unit test.

## Listing 16. TestContactService

```
import unittest
from contacts import ContactService
from dbscript import *

class TestContactService(unittest.TestCase):

    def setUp(self):
        self.cs = ContactService()
        try:
            drop_table()
        except:
            print "unable to drop contact table"
        try:
            create_table()
        except:
            print "unable to create contact table"

    def testAdd(self):
        clear_table()
        cs = self.cs
        cs.addContact({"name":"Richard",
                       "phone":"5205551212",
                       "email":"rick@rick.com"
                      })
        list = cs.listContacts()
        print list
        found = False
```

```
        for cdict in list:
            if cdict["name"]=="Richard": found = True
        self.assertTrue(found)

    def testUpdate(self):
        cs = self.cs
        insert_test_data()
        cs.updateContact(
            {"name":"Richard",
                       "phone":"5205551212",
                       "email":"rick@rick.com",
                       "id":1})

        list = cs.listContacts()
        print list
        found = 0
        for cdict in list:
            if cdict["name"]=="Richard": found +=1
        self.assertTrue(found==1)


    def testRemove(self):
        cs = self.cs
        insert_test_data()
        cs.removeContact(
            {"name":"Richard",
                       "phone":"5205551212",
                       "email":"rick@rick.com",
                       "id":1})

        list = cs.listContacts()
        print list
        found = 0
        for cdict in list:
            if cdict["name"]=="Richard": found +=1
        self.assertTrue(found==0)




if __name__ == '__main__':
    unittest.main()
```

The `dbscript.py` in Listing 17 can build either a MySQLdb contacts table or a sqlite3 contacts table.

## Listing 17. Dbscript, create, drop, populate contacts

```
use_mysql = True

if use_mysql:
    import MySQLdb as db_api
else:
    import sqlite3 as db_api

db_url = "/tmp/contacts"


create_table_sql = """
create table contact (
     id INTEGER %s PRIMARY KEY,
     name VARCHAR(50),
     phone VARCHAR(50),
     email VARCHAR(50));
"""
```

```
if use_mysql:
    create_table_sql = create_table_sql % ("AUTO_INCREMENT",)
else:
    create_table_sql = create_table_sql % ("",)


def run_script(func):
    if use_mysql:
        connection =  db_api.connect(passwd="mypassword", db="contactdb", user="root")
    else:
        connection =  db_api.connect(db_url)

    cursor = connection.cursor()
    try:
        func(cursor)
        connection.commit()
        cursor.close()
    finally:
        connection.close()

def create_table():
    def ct(cursor):
        cursor.execute(create_table_sql)

    run_script(ct)

def drop_table():
    def dt(cursor):
        cursor.execute("drop table contact;")
    run_script(dt)

def clear_table():
    def dt(cursor):
        cursor.execute("delete from contact;")
    run_script(dt)

def insert_test_data():
    def itd(cursor):
        if use_mysql:
            cursor.execute("insert into contact (id, name, phone, email) values (NULL,
               'Bob', '5', 'b@b.com');")
            cursor.execute("insert into contact (id, name, phone, email) values (NULL,
                'Rick', '5', 'b@b.com');")
            cursor.execute("insert into contact (id, name, phone, email) values (NULL,
                'Sam', '5', 'b@b.com');")
        else:
            cursor.executescript("""
    insert into contact (id, name, phone, email) values (NULL, "Bob", "5", "b@b.com");
    insert into contact (id, name, phone, email) values (NULL, "Rick", "5", "b@b.com");
    insert into contact (id, name, phone, email) values (NULL, "Sam", "5", "b@b.com");
           """)

    run_script(itd)
```

Essentially, the `dbscript` creates and drops the contacts table and populates it with test data that the unit test uses. After you've completed the JSON-RPC service, you can install it to be served by Apache HTTPD by adding the code shown in Listing 18 to the httpd.conf file.

## Listing 18. /etc/apache2/httpd.conf

```
Alias /services "/home/rick/services"

<Location /services/>
    AddHandler mod_python .py
    PythonHandler jsonrpc
</Location>
```

Remember, after making changes to the service, you need to restart it, as shown in Listing 19.

## Listing 19. Restarting Apache2 to pick up changes to mod_python

```
$sudo /etc/init.d/apache2 restart
```

You can get nasty recursion errors when running JSON-RPC proxies in Pyjamas. To help debug errors, I used the JSON-RPC stand-alone client lib, as shown in Listing 20.

## Listing 20. Python JSON-RPC client

```
from jsonrpc import ServiceProxy, JSONRPCException

cs = ServiceProxy("http://localhost/services/contacts.py")

if cs.test()=="test":
    print "connected"

try:
    cs.addContact(
        {"name":"Larry Wall",
         "phone":"5551212",
         "email":"rick@rick.com"})

except Exception, e:
    print e.error
    print `e.error`
```

The previous step is an important one in testing and debugging. Pyjamas development is a bit nascent, so it's good to have another way to test the JSON-RPC from another source.

The example just changes the `ContactService` to use a JSONProxy. The JSONProxy is Pyjamas client support for JSON-RPC. You can create a proxy object to the service you just wrote, as demonstrated by the `ContactsJSONProxy` in Listing 21. The return objects from the JSON service are returned asynchronously. Thus, when you make calls on the JSON proxy you pass an instance of `ContactService`, which implements the `onRemoteResponse` to get the responses from the service asynchronously.

## Listing 21. JSONRPC-ized contact listing

```
from pyjamas.JSONService import JSONProxy
...
class Contact:
    def __init__(self, name="", email="", phone="", id=None):
        self.name = name
        self.email = email
        self.phone = phone
        self.id = id
    def to_dict(self):
        return {"name":self.name, "email":self.email,
```

```
                "phone":self.phone, "id":self.id}


class ContactsJSONProxy(JSONProxy):
    def __init__(self):
        JSONProxy.__init__(self, "/services/contacts.py",
                           ["addContact", "removeContact",
                            "updateContact", "listContacts","test"])



class ContactService:
    def __init__(self, callback):
        self.callback = callback
        self.proxy = ContactsJSONProxy()

    def test(self):
        self.proxy.test(self)

    def addContact(self, contact):
        self.callback.showStatus("Add contact called")
        self.proxy.addContact(contact.to_dict(), self)

    def updateContact(self, contact):
        self.callback.showStatus("Update contact was called")
        self.proxy.updateContact(contact.to_dict(), self)


    def removeContact(self, contact):
        self.callback.showStatus("Remove contact was called")
        self.proxy.removeContact(contact.to_dict(), self)


    def listContacts(self):
        self.proxy.listContacts(self)


    def onRemoteResponse(self, response, request_info):
        if request_info.method == "addContact":
            self.callback.service_eventAddContactSuccessful()
        elif request_info.method == "updateContact":
            self.callback.service_eventUpdateContactSuccessful()
        elif request_info.method == "listContacts":
            def toContact(x):
                return Contact(x["name"], x["email"], x["phone"], x["id"])
            contacts = map(toContact, response)
            self.callback.service_eventListRetrievedFromService(contacts)
        elif request_info.method == "removeContact":
            self.callback.service_eventRemoveContactSuccessful()
        else:
            self.callback.showStatus(""" REQ METHOD = %s RESP %s """ %
                 (request_info.method,response))

    def onRemoteError(self, code, errobj, request_info):
        message = errobj['message']
        if code != 0:
            self.callback.showStatus("HTTP error %d: %s" % (code, message))
        else:
            json_code = errobj['code']
            self.callback.showStatus("JSONRPC Error %s: %s" % (json_code, message))
```

The rest of the client code is fairly similar to the way it was before, there were only some cosmetic changes. The amazing part is that the client that uses the real remote RPC service is not much different than the version that uses the stand-alone version of the service. This lets you quickly

develop the GUI and then just plug in the JSON-RPC service, which are developed and debugged separately.

## Summary

In this first part of the "Introduction to Pyjamas" series, you explored the history and vision behind Pyjamas. You also learned how to create a Pyjamas-based application using Pyjamas, mod_python, and Python JSON-RPC. Stay tuned for Part 2 of this series, which will explain how to build custom Pyjamas components.

## Acknowledgements

Special thanks to Luke Kenneth Casson Leighton for reviewing this article and giving vital feedback. He also helped get the examples running, and gave advice on how to debug the applications.

# Downloads

| Description | Name | Size |
| --- | --- | --- |
| Sample code for contact management application | pyjamas.zip | 2425KB |

## Resources

- Learn more about Pyjamas and why you should use it.
- Google Web Toolkit: For building and optimizing complex browser-based applications.
- Use the Pyjamas API documents.
- *Why we need a Common Free Software Applications Interface*, by Luke Kenneth Casson Leighton, is a good introduction and reference for Pyjamas.
- Learn more about the Python programming language.
- IBM technical events and webcasts: Stay current with developerWorks' technical events and webcasts.
- My developerWorks: Personalize your developerWorks experience.
- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

## About the author

**Rick Hightower**

Rick Hightower, a software developer with 20 years of experience writing software, is a frequent writer for developerWorks. He is the chief technology officer for Mammatus Inc., a San Francisco Bay Area training and consulting company. Rick is co-author of the popular book *Java Tools for Extreme Programming*, and is an author of *Programming the Java API with Python*. He is on the editorial board for Java Developer's Journal, and is a frequent contributor to the Java and Groovy topics on DZone.