



Mobile Technologies in EPICS-2

A report produced as part of the North East regional collaboration for personalised, work-based, and life-long learning (EPICS-2)

<http://www.epics.ac.uk>

Report author: Paul Horner

Key contributors:

Paul Horner Work Package Lead

Simon Cotterill Project Manager

Full EPICS-2 project report and acknowledgements:

<http://www.epics.ac.uk/report>

EPICS

NE Regional ePortfolios & PDP
Collaboration - www.epics.ac.uk

JISC

Contents

1. Summary	3
1.1 Introduction	3
1.2 Aims.....	3
1.3 Outcomes.....	3
1.4 Methodology.....	3
2. Previous Work – A Pilot Project	3
2.1 Background	3
2.2 A Browser Based-approach.....	4
2.3 Conclusions from the Pilot.....	4
3. An Asynchronous Approach.....	4
3.1 Offline Access.....	4
3.2 Proprietary Software.....	4
3.3 Conclusions	5
4. A Web Framework	5
4.1 Core expertise	5
4.2 A Proof of Concept using Django	5
4.2 A script to run Django on a PDA.....	6
4.3 Barriers to this approach	6
4.4 Conclusions	7
5. Text Messaging	7
5.1 A different approach.....	7
5.2 JanetTxt.....	8
5.3 A Pilot study	8
6. Conclusions	9
6.1 Background	9
6.2 Our Approaches to this problem	9
6.3 Future Developments	9
6.4 Lessons Learnt.....	10
6.5 Recommendations	10
Appendix 1	11
1. Background	11
2. Sending messages (the SOAP Client)	11
3. Receiving Messages (the SOAP Callback Server)	14
4. Conclusions	17

1. Summary

1.1 Introduction

The EPICS-2 Project¹ investigated the use of ePortfolios in personalised, work-based and lifelong learning. The main aim of the project was to significantly increase the support for these learning types in the North East of England and beyond. Central to the idea of personalised learning was providing access to ePortfolios through mobile telephones and Personal Digital Assistants (PDAs),² as this would allow learners to reflect upon their learning while outside of the traditional classroom, or even campus environment.

In the EPICS-2 Project, we wanted to build upon the investigative work previously conducted by Newcastle University in providing access to ePortfolios through mobile devices. Our work in this area had largely been focussed on providing web-based access to the standard university ePortfolio via a PDA's inbuilt web browser. However, in EPICS-2, we wanted to take this one step further, to allow students to update their ePortfolio even when a reliable internet connection was not available.

1.2 Aims

In this study we intended to investigate the methods in which ePortfolios could be supported by mobile technologies. We wanted to find a method which allowed students to update their ePortfolio, without having to rely on an internet connection.

1.3 Outcomes

We developed a tool using SMS text messages that allows undergraduates at Newcastle University to add blog entries through their mobile telephone. We have also produced a technical report which should aid other institutions to implement a similar system using the JanetTxt service.

1.4 Methodology

Following a pilot project at James Cook University Hospital in Middlesbrough, we researched the many options available to us. Using several of these techniques we crafted basic models and tested these to see which would be most practical. We eventually developed a workable tool using SMS text messages (through the JanetTxt service), which was made available through the undergraduate ePortfolio at Newcastle University.

2. Previous Work – A Pilot Project

2.1 Background

A pilot project funded by the Centre for Excellence in Teaching and Learning for Healthcare in the North East (CETL4HEALTH), provided PDAs to medical students at the James Cook University Hospital in Middlesbrough. This pilot ran from 2005-2007 and impacted upon three cohorts of fifth year students and two of third years. The Hospital had been running a successful 'Hospital at Night' scheme, which used PDAs to contact doctors on call during nightshift to inform them of whereabouts in the hospital they were required to be. This scheme used an extensive wireless network that covered the majority of the hospital. This wireless network, coupled with the

¹ <http://www.epics.ac.uk>

² PDA – Personal Digital Assistant

hospital's experience of dealing with mobile technologies provided the perfect opportunity to test the effectiveness of using PDAs with undergraduate students.

2.2 A Browser Based-approach

A mobile version of the ePortfolio was produced by developing a custom stylesheet that would be accessed by mobile devices, and a browser-recognition script that could easily remove superfluous html styling to make the ePortfolio fit more comfortably into a PDA's tiny screen resolution. We added some additional functionality to the ePortfolio, to replicate the paper logbooks that the students traditionally had to complete during their time in the hospital. The paper logbooks contain reflections, tick-boxes and reviews by peers and hospital staff (who have to sign to say that they have seen a student in a particular scenario). We managed to replicate the paper logbook, while ensuring that the benefits of an electronic version were maintained. One popular addition to the system was a facility that captured signatures, which negated the requirement for all hospital staff to have a username and password.

In addition to the web connection, the PDAs also included software that was designed to replace the textbooks that traditionally medical students have to carry on their persons, including a medical dictionary and the British National Formulary (BNF). This additional software was designed to serve as a hook, to increase usage of the PDAs thereby encouraging students to use their PDAs to complete their ePortfolios.

2.3 Conclusions from the Pilot

As a proof of concept this was a success. We proved that PDAs can support medical education, and demonstrated that it could be particularly helpful in supporting students away from campus.

The success of this project meant that we wanted to extend the usage of mobile technologies across the curriculum. However, there were a number of limitations that meant that the James Cook model could not be made available elsewhere. The key factor was the reliance on a wireless network. Very few locations where students would be located off-campus had wireless networks, and of those that did, none had as extensive or reliable a network as James Cook.

3. An Asynchronous Approach

3.1 Offline Access

The solution to this issue was to provide an asynchronous connection to the ePortfolio. The student would be able to complete their ePortfolio online or offline, and if completing offline could synchronise with their online portfolio when they connected to the internet. In EPICS-2, we attempted to provide such a service to undergraduate students at Newcastle University.

3.2 Proprietary Software

A number of software packages are available to allow asynchronous connections to mobile devices. Generally, these consist of a form-designer, some kind of tool to add these forms to a mobile device, and a synchronisation server to put data recorded on the mobile devices into the online database.

We looked into two of these packages. The first, mForms, was developed in the UK and is largely Windows based. The form-building interface is similar to Microsoft Access, and synchronisation uses an Access (or MSSQL) Database. At the time, our internal infrastructure supported unix servers and the ePortfolio database is MySQL. This provided a major barrier in terms of support. There were also some difficulties with replicating the complexities of the ePortfolio system within the forms created. The most likely cause of this was our own inexperience of using the software rather than a limitation of the software itself.

An alternative solution to Mforms, called Go-DB, was also investigated. This provided a less-Windows focussed approach, and synchronisation could be conducted using a dedicated server or through web services. The interface to this was very similar to that of mForms, and it provided similar issues while attempting to replicate even the simplest components of the ePortfolio.

3.3 Conclusions

The reliance on a proprietary application to provide this software to learners also provided a barrier. The ongoing support and licensing costs were a factor, as was the ongoing maintenance requirement. The software would need to be installed and maintained on all PDAs, and we felt that this would impact too much on our time. These tools would ultimately provide a quite static interface to the ePortfolio. Any changes to the portfolio structure or functionality would need to be replicated in these forms and reinstalling this on multiple devices would be very time consuming. As such, we decided that this was not the best solution for a mobile interface to ePET, although we did note that we could see some excellent use-cases, particularly if we were looking at a small-scale pilot, interacting with just one smaller ePortfolio component.

4. A Web Framework

4.1 Core expertise

Our core area of expertise is in the development of online systems, and as such we decided to investigate how this could be used to serve an off-line ePortfolio. This would mean the development of a new version of ePET (or a small subset of ePET components) in a web framework, which would be installed on a PDA/mobile phone and run as a standalone (offline) tool and synchronised with the online version when a network connection is present.

A basic server would serve the web pages (many frameworks come with their own basic server), a small database would store the data (SQLite would be the obvious choice), and a PDA version of the system would be installed onto SD cards so that we don't need to worry about disc space. To synchronise an unobtrusive HTTP request would be used to determine if the site is online/offline and a web-service would send new data up to the server and retrieve new data from the online system (when a connection is present).

4.2 A Proof of Concept using Django

We began by looking into Django, a Python-based web framework, which comes with its own webserver. A quick search on Google revealed that a small number of people have already managed

to get Django working on mobile devices, including PDAs³ and the iPhone⁴. In terms of the PDA, there is a Mobile version of Python⁵ available, which made installing Django surprisingly easy.

Django also ships with support for SQLite, a lightweight SQL database system. The installation of this on a PDA was more complicated, but I managed to install a pre-compiled .dll file used by the .net instance of SQLite⁶. Connecting to this database with Django is very straightforward, although the database files had to be created outside of Django. However, once created, it was simply a case of pointing the Django settings file to the new database file and Django could connect to it.

A very simple tool was created in Django. This allowed the user to create a blog entry offline and synchronise it with their online ePortfolio blog. On synchronisation it also retrieved a copy of their existing categories from their online blog, which meant that the learner could assign these while offline, while also demonstrating a two-way synchronisation process. The development of this tool was completed in a desktop environment, and then copied to the PDA. No changes were required when porting this to the new environment, except that the database had to be built manually and a separate python script was written to run the webserver (as localhost rather than as an ip).

4.2 A script to run Django on a PDA

Running Django on a PDA is very different to running it on a server or PC. The mobile version of Python is quite different, and it is relatively difficult to run Python through the command line. The following script was used to actually run the server. The PDA user would simply click on this file, which would open the Python interpreter, which in turn would start the webserver (running at <http://localhost:8000>), and open the browser, pointing to that local address.

```
import sys, socket, webbrowser

#run the webserver at http://localhost:8000
sys.argv.append("runserver")
sys.argv.append("localhost:8000")
sys.argv.append("--noreload")

#open the browser
webbrowser.open("http://localhost:8000/")

#start the django webserver
from django.core.management import execute_manager
import settings # Assumed to be in the same directory.
execute_manager(settings)
```

4.3 Barriers to this approach

As a proof of concept it worked, but in practice there were three quite fundamental issues that would have created a significant barrier to using this technology. The first issue was that the Python script always appears in the background. This in itself is probably not a big problem, but the user could easily close it down and it does need to be there in order to work properly, and returns a DNS error

³ <http://marxy.org/2006/10/django-on-windows-mobile-5.html>

⁴ <http://blog.wired.com/monkeybites/2007/08/running-django-.html>

⁵ <http://pythonce.sourceforge.net/Wikka/HomePage>

⁶ <http://sqlite.phxsoftware.com/>

if the user closes it.

The second issue is that installing onto an SD card did not work as intended, and when moved to a different PDA it didn't run the python scripts. This is because changes were required in the PDA registry to acknowledge Python as a program. In order to run from an SD card there would need to be an executable file that runs Python first, or an installer that creates all the right files and puts them in the right places.

The third, and by far the biggest, issue was that it used an incredible amount of overhead (for a PDA). If the PDA was running any other program than Python and the webserver, it started to run very slowly. Sometimes the server failed to start when additional programs were running prior to startup. It uses so much overhead that running any other browser than Internet Explorer (Opera, Minimo) used up too much memory and actually crashed the browser. The main drawback with this approach was that it did not use a technology native to the PDA. It essentially meant that the PDA was being used as a webserver and the processing power of these devices mean that this really is above and beyond what can be expected of them.

4.4 Conclusions

Developing a web application on a PDA was a useful experience, and although at this time we found there to be too many barriers to implementing it, the advances in mobile technologies, coupled with the advances in web frameworks, mean that this might be a possible solution in the longer-term. It provided us with far more flexibility than any of the form-building tools, and we managed to demonstrate a real, two-way synchronisation between a mobile device and a webserver. Having ownership of the development environment will mean that existing web services and emerging interoperability standards can be used to synchronise data from handheld devices. This will be a massive leap forward in the future, but we will have to wait for the technology and standards to catch up.

5. Text Messaging

5.1 A different approach

Following our work with Django, our opinion on what we actually wanted to achieve shifted slightly. At this time we had received a lot more feedback from students following the James Cook pilot. We discovered that students don't want to use PDAs for reflection, preferring a more 'tick-box' oriented approach.

Following this feedback, we decided to take a step back and look at other mobile applications. We found that, particularly in the web2.0 world, that two approaches were taken. The first was to provide a version of the website that could be viewed natively on a mobile device. We had already provided this through the James Cook pilot and felt that the fundamental issue of access had not yet been addressed in order to make this more widely available. The second approach was to allow content to be added by text message.

After much deliberation, we came to the conclusion that this approach could be a real step forward for the ePortfolio. It was a very simple approach, certainly far simpler than writing an offline web application and server, but it did cover many of the key elements of what we wanted from a mobile ePortfolio.

1. We would not have to provide students with hardware. Most students have their own mobile phones and the vast majority of modern mobile phone contracts offer a large number of free text messages. We did not envisage that students would see the cost of sending a text message as a barrier, as long as they could see the benefit of using the service.
2. Most mobile phone users are familiar with text messaging and therefore we believed that there would be very little training required. We also expected that students would be happier writing narrative using a text keypad than using the PDA stylus.
3. Text messages by their very nature are asynchronous. If a message fails it is stored and can be sent later.
4. We would not have to install anything on the PDAs/Mobile phones.

5.2 JanetTxt

A service offered by Janet, called JanetTxt, provided us with the facility to send and receive text messages. The cost of sending text messages was a little over 4p, but the main benefit of the JanetTxt service was the SOAP API which allowed us to receive text messages directly into the ePortfolio.

Developing the interface using the SOAP APIs was quite a complex process, and our choice of technology did not suit the manner in which PageOne/JanetTxt have implemented their SOAP service⁷. However, this service was made available to students in January 2009. The students simply text their message to 07624 810 331, and their message is automatically added into their ePortfolio blog. For security reasons these blog entries are private, so that only the student can see them until they have been amended.

5.3 A Pilot study

It is a very simple, small-scale pilot, but it has been made available and advertised to all students who actively use their Newcastle undergraduate ePortfolio (c. xxxx students).

Initial feedback was actually quite mixed. We ran a focus group with students from Speech Therapy, who we believed would greatly benefit from this service. They are away from campus for a long period, on work-based placements in health centres around the region, and they are active blog users, using this reflective practice to produce evidence towards assessment. There was a general uneasiness about the service, with students being concerned that using their personal mobile phones crossed a line between 'work' and 'home'. There was also some concern about professionalism when using this service. The main reason for using the service was to record reflections about interactions with real patients. However, many students worried how it would be perceived if the student was sending text messages during work time. If a patient or colleague was to see them doing this, they worried that this would appear unprofessional.

Some students noted that the price may be an issue. Although every student had a mobile phone, and more than 90% of students had a contract which provided free text messages, some students felt that they should not have to cover the costs of sending messages. We will have to review the feedback at the end of the pilot project to see how much of a barrier costs actually were in practice.

⁷ For a technical overview of our work with JanetTxt, see appendix 1.

Further feedback will be provided at the end of this pilot project, and a final report on this will be made available on the EPICS website – <http://www.epics.ac.uk>.

6. Conclusions

6.1 Background

Mobile technologies have a place in education. If used correctly, they can greatly enhance the student experience, and when coupled with ePortfolios, can encourage and improve upon reflective practice. In EPICS-2, we investigated several methods of using mobile technologies to support ePortfolios, and we have tested two very different approaches with real student.

6.2 Our Approaches to this problem

The first approach is simply to provide an interface to the ePortfolio that is easy to use in a mobile device. This is relatively simple to do, and requires little more than a custom stylesheet and a browser-recognition script. However, it is very difficult to do correctly, and we have to be careful to ensure that the mobile device is used in the way it was intended to be used. Writing reflections on these devices is very difficult, but checkboxes and selecting pre-defined answers works very well, which means that the content and functionality of the mobile portfolio may need to differ greatly from the standard ePortfolio.

In contrast our second approach does not use web technologies, and is a one-way process. The learner simply sends a text message to a particular number, and that message gets added into their ePortfolio. The minor issues with this were a loss of feedback from the website, limitations in the text being added to the blog records, and the student having to cover the cost of the text message. However, these issues were balanced by the additional benefits of this route. The ongoing costs are massively reduced because we don't have to install any software, and because we don't have to provide any hardware to the learners. Support is also minimised because the vast majority of users are already familiar with the concept.

6.3 Future Developments

As a proof of concept we have demonstrated that mobile technologies can support ePortfolios for learning, and with our two approaches we have provided our students with new and simple ways to add content to their ePortfolios. As technology advances the options available to us will increase and improve, and in the short-to-medium term, mobile technologies will have an even greater bearing on teaching and learning.

The future of mobile technologies in learning is limited only by technological and educational innovations. We have covered in detail the benefits of using mobile technologies, and these benefits will be intensified as education becomes more global, and as more emphasis is placed on the individual. Advances in technology will mean that in the future it will be much easier to develop tools for mobile devices. New technologies such as Google's Android, and the iPhone are already beginning to realise that, and as the price of 3G mobile broadband gradually reduces, and the coverage gradually increases we may see a time when an online version of the mobile ePortfolio will be the easiest and most appropriate solution.

At Newcastle, we are already starting to build upon our text messaging service, and are starting to look into ways of making this a two-way communicative process, sending text messages to students through the ePortfolio and other online systems. In the short-term this will enable us to communicate details of timetable changes to students, or details of lift outages to staff and students with mobility issues. As time progresses, we may provide ways of promoting group discussion and community learning using text messaging through the ePortfolio, and we are also investigating ways to use MMS messaging to transfer multimedia as well as text.

6.4 Lessons Learnt

Providing access to an ePortfolio via a mobile device is not a trivial task. We explored many avenues before finally discovering something that worked for us. This was a useful exercise, but unfortunately due to our particular, unique requirements, some of these avenues turned out to be cul-de-sacs (particularly the form builder tools and web framework).

There must be consultation with students at all stages in the process. We found that students were very receptive to the idea of using mobile technologies, but in practice there was some concern that it could be viewed as 'gimmicky' or even unprofessional. Consultation with students from the outset could have helped dispel these myths.

One key lesson learnt from the James Cook pilot was that PDAs do not have a very long shelf life. If investing large amounts of resources in purchasing these devices, you need to assess the longevity. Within three years the James Cook PDAs went from being cutting edge technology to being obsolete, and practically worthless. The majority of new mobile telephones are being produced with 3g and web browsers as standard, and as such the future may allow us to use the students' own telephones rather than investing in expensive hardware.

6.5 Recommendations

Text messaging worked for Newcastle University, but that might not be the case for other institutions. The easiest solution is undoubtedly to provide a mobile interface to the existing ePortfolio. This would be particularly useful in urban surroundings where 3g mobile broadband is widely available, and there are a large number of available wifi hotspots. In the North East we have a large rural base, where 3g access isn't always an option, and we also found that the majority of off-campus locations where students were on placements did not have wireless networking. This was the main driver behind using an off-line approach.

Appendix 1

Technical report on JanetTxt SOAP APIs

1. Background

In EPICS-2, we needed to deploy JanetTxt through the existing ePET ePortfolio system. This meant that we used the JanetTxt service in a slightly different way to most other institutions. Our work used the JanetTxt SOAP (Simple Object Access Protocol) API (Application Programming Interface). SOAP uses XML to transfer data over the HTTP protocol, in much the same way that data is transferred when accessing websites. As a way of accessing and transferring data SOAP is extremely powerful, although as we found when trying to implement the JanetTxt SOAP API, it can also be extremely complicated and if the server and client are not implemented in exactly the right way, it can be very difficult to communicate using SOAP.

2. Sending messages (the SOAP Client)

Although EPICS-2 focussed primarily on sending text messages into the ePortfolio through JanetTxt, we did investigate the use of the API to send text messages. The process for sending messages is relatively straightforward. You simply login, by sending your username and password and then send your message (and an array of numbers to send the message to). In practice this process was not quite so simplistic.

The ePet ePortfolio system is built using Zope, a Python-based web development framework. Zope allows us to call Python scripts from the server's file system through our web application (external methods). There are two main SOAP libraries for Python – SOAPpy and ZSI. Even though SOAPpy is now deprecated, and much of the functionality is being incorporated into ZSI, because we have had some success with SOAPpy in the past we decided to use this as a first port of call. SOAPpy is a relatively lightweight library, but it can interface with WSDL (Web Service Definition Language) files or the actual SOAP objects themselves.

Below is a simple example of a SOAPpy call.

```
from SOAPpy import SOAPProxy, WSDL
def soap_eg():
    wsdlFile = 'http://soap.server.com/wsdl/myWSDLFile.wsdl'
    server = WSDL.Proxy(wsdlFile)
    return server.my_soap_method()
```

This SOAP call simply calls a function called my_soap_method at the url specified. It returns a copy of the XML response from the server. As you can see, this is very straightforward and relatively easy to get to grips with. However, when attempting to integrate this with JanetTxt, we came across a major problem in the way that PageOne (the company who developed JanetTxt and provide the API) have implemented SOAP, and the limitations of the SOAPpy library.

A very basic example is the login method. This simply uses a function called 'ovLogin' that requires an object called 'request', that contains an array of objects – 'user-id' (our JanetTxt username) and

'pwd' (our JanetTxt password). However, PageOne's reliance on namespaces means that the 'ovLogin' object needs to be passed as a 'loginRequest' method. Unfortunately, SOAPpy could not handle this and the returned error stated that it needed to be an ovLogin. Passing an ovLogin, stated that it needed to be a loginRequest.

ZSI was similarly confusing, and although it probably could handle this request, the documentation available is very limited and the online examples are based on much more simplistic use-cases than JanetTxt.

The alternative adopted was to simply send the XML as a HTTP Request. This was really a backwards step, but it did allow us to take full control over the information we were sending to the JanetTxt SOAP service. The Python HTTPLIB Library allows us to send any data as the request, and there is also extra flexibility in the headers we send. By manipulating this information we can easily formulate a SOAP request in exactly the same format that is being requested. Where this approach falls short is that every request has to be written manually. It does not use SOAP as a way of calling functions on a remote server, but instead simply sends the XML that the server is expecting.

An example SOAP request using HTTPLIB

```
import httpplib
def http_ping():
    conn = httpplib.HTTPSConnection("soap.ovens.com")
    conn.connect()

    soapstr = """<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<ping SOAP-ENC:root="1"></ping>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>"""
    #soap headers
    conn.putrequest("POST", "/webservices/soap")
    conn.putheader("Content-Type", "text/xml; charset=utf-8")
    conn.putheader("Content-Length", str(len(soapstr)))
    conn.putheader("SOAPAction", "ping")
    conn.endheaders()

    conn.send(soapstr)
    theResponse = conn.getresponse()
    theXml = theResponse.read()

    return theXml
```

This actual script is used to check that the server is available and that the current session is available. The returned XML needs to be parsed to return the actual variables that make up the response. Using a standard SOAP library would return these variables as objects. However, using HTTPLIB means that the returned information is simply the XML response, which needed to be manipulated.

Mobile Technologies in EPICS-2

A simple function was written using the Python SAX library which rendered the contents of the SOAP responses as a Python dictionary.

The XML parser –

```
from xml.sax import parseString
from xml.sax.handler import ContentHandler
import string

class MyXMLHandler(ContentHandler):
    element = ''
    return_dictionary={}
    temp_var = ''
    this_element = ''
    this_content = ''

    def startElement(self, name, attrs):
        self.this_element = name

    def endElement(self, name):
        if self.this_content <> '':
            self.return_dictionary[self.this_element] = self.this_content

            self.this_element = ''

    def characters(self, content):
        if content.strip():
            self.this_content = content.strip()
        else:
            self.this_content = ''

def soap_xml_parse(data):
    message = data.strip()
    handler = MyXMLHandler()
    parseString(message, handler)
    return handler.return_dictionary
```

The response from the SOAP call was in the following format –

```
<?xml version="1.0" encoding="utf-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Header>
  <m:header xmlns:m="http://schemas.oventus.com/">
    <ovHeader>
      <session-id>SOAP_33A7A50A5C8EF69BE70F786AEB944BCA</session-id>
    </ovHeader>
  </m:header>
</SOAP-ENV:Header>
<SOAP-ENV:Body />
</SOAP-ENV:Envelope>
```

The soap_xml_parse function simply returned a dictionary object

```
{'session-id': 'SOAP_33A7A50A5C8EF69BE70F786AEB944BCA'}
```

This was a very simplistic parser, and there are some limitations. It does not check for attributes on any XML element, although this can be extended by manipulating the attrs object in startElement. It also overwrites any previous entries in the dictionary if a new element exists with the same name. We have extended it to cover this eventuality, but for the purposes of JanetTxt it did not seem to cause any problems.

The use of SOAP to send text messages should be relatively straightforward. It should require some kind of authentication and authorisation, but after that initial login stage should simply be a case of posting the numbers you want to send the message to, alongside the message you want to send. The basic principle of the JanetTxt API does allow this to happen. However, in practice this did not work for us quite as easily as we had hoped. A REST interface to this API would have been far easier to use and would work easily regardless of platform or scripting language. A REST approach is becoming more common for web services and we would like to see JanetTxt implement a REST model.

3. Receiving Messages (the SOAP Callback Server)

The key functionality for EPICS-2 was to receive text messages from our students. The ePet portfolio system is written in Zope, and as mentioned in the section above, sending SOAP requests from Zope can be done using Python external methods. However, receiving SOAP calls is not quite as easy. The Zope interface expects HTTP requests to be 'normal' HTTP requests, and does not allow us to access the actual SOAP request object. Zope is incredibly efficient at handling XML-RPC requests, and effectively all Zope methods can be called using XML-RPC.

We are currently in the process of migrating some of our core services into Django, and as such we investigated hosting the SOAP server in a Django-based system. However, this proved to be less successful than we had hoped. Django contains excellent support for REST-based web services but because the web framework is still quite new, and despite rapidly growing, unfortunately SOAP support remains in its infancy. It is possible to make a Django method handle a SOAP request, but there are some limitations to this facility. For example, access to the SOAP-ENV:header is not currently possible, and similar problems were experienced when setting the correct response as we found when writing the SOAP requests using Python's SOAPpy library.

As an alternative to Django and Zope, we looked at another Python web framework called CherryPy. This is a very lightweight system that allows you to publish web pages in simple python script. The lightweight nature of this system meant that it was possible to write a very basic script that simply checked the XML data and headers that formed a HTTP request, parse that XML, and send back a response in exactly the format we wanted to use. Similarly to the SOAP requests, this actually involved writing our own XML. Again, this was not the preferred option, but due to the nature of the difficulties we had experienced when using the Python SOAP libraries, and the time constraints placed upon us, this was deemed to be the timeliest option.

Mobile Technologies in EPICS-2

An example of a CherryPy server is shown below. This is similar to the option we adopted, with some minor changes.

```
import cherrypy

# Configure the CherryPy Server
cherrypy.config.update({'server.socket_host': '127.0.0.1',
                        'server.socket_port': 9000,
                        'response.timeout': 6000,
                        'server.thread_pool': 10,
                        })

# JanetTxt SOAP server
class JanetTxtServer(object):
    def index(self):
        #the request object
        xmlBody = cherrypy.request.body.read()

        # respond with XML
        retXml = """<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope>
  <SOAP-ENV:Body>
    <m:replyResponse xmlns:m="http://schemas.ovens.com/">
      <ovAcknowledgeCallBack>
        <response><acknowledge>123</acknowledge></response>
      </ovAcknowledgeCallBack>
    </m:replyResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>"""

        cherrypy.response.headers['Content-Type'] = 'text/xml;
charset=utf-8'
        cherrypy.response.headers['Content-length'] = str(len(retXml))
        return retXml
    index.exposed = True

# Basic root page for this server
class Root(object):
    janettxt = JanetTxtServer()
    def index(self):
        my_html = """<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"><html
xmlns="http://www.w3.org/1999/xhtml">
<head><title>EPICS-2 SOAP Server</title></head>
<body>
  <h1>EPICS-2 SOAP Server</h1>
  <p>This is the epics SOAP server</p>
</body></html>"""
        return my_html
    index.exposed = True

# serve the pages
cherrypy.quickstart(Root())
root = Root()
root.janettxt = JanetTxtServer()
```

The main issue with this option was parsing the XML into something that Python could understand. This is far more complicated than simply writing XML. The method we adopted to implement this used a Python library called SAX. We have previously used this for parsing XML in ePortfolio interoperability standards, and for quite straightforward XML it works very well. The SAX library reads through the XML element by element. You tell it what to do when an XML element is opened, when it is closed, and what to do with the content of that element.

Below is an example of a Python SAX Parser. Again, this is similar to the script used, but with some minor changes. For example, this script does not check any attributes on the XML elements.

```
from xml.sax import parseString
from xml.sax.handler import ContentHandler

# parse the XML
class MyXMLHandler(ContentHandler):
    return_dictionary={}
    this_content = ''
    this_element = ''

    # when I find a new XML element
    def startElement(self, name, attrs):
        self.this_element = name

    # when an XML element is closed
    def endElement(self, name):
        if self.this_content <> '':
            # add this to the return dictionary
            self.return_dictionary[self.this_element] = self.this_content
            self.this_element = ''

    # do something with the strings
    def characters(self, content):
        self.this_content = content.strip()

# call the XML parser
def EPICS_parser(data):
    mymessage = data.strip()
    handler = MyXMLHandler()
    parseString(mymessage, handler)
    return handler.return_dictionary
```

The parsed XML is returned as a Python dictionary.

```
{
  'session_id': 'ThisIsMySessionId',
  'recipient': '077*****',
  'dateTimeOfReq': '2008-12-01T09:30:47.0Z',
  'dateTimeOfResp': '2008-12-01T09:30:47.0Z',
  'message': 'This is my Text message'
}
```

To put this information into the database is quite trivial. We simply match the recipient telephone number with the university records, and insert this message (with the corresponding time stamp) into the student's ePortfolio blog. Some additional information is added to make this limited record fit into the blog, and some security measures are added to make sure that text message entries are not made public until the student edits them online.

4. Conclusions

The methods we used were not actually the methods we initially wanted to use, but due to the constraints placed upon us by the available technology we had to find some slightly quirky solutions to the issues presented. Our final code works, and provides a workable and easy to use solution to the problem of mobile integration. Python as a programming language is extremely flexible, and although the support for SOAP web frameworks is less complete than in other languages, the flexibility meant that solutions could be sought from elsewhere within the Python community.