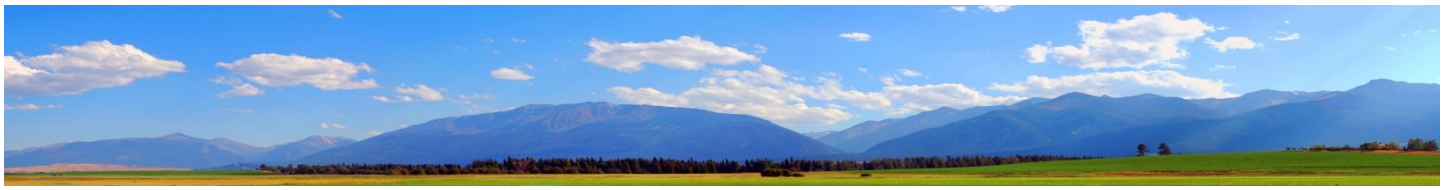


MultiValue Bookstore Sample
Python Examples Volume 1

Brian Leach Consulting Limited

http://www.brianleach.co.uk

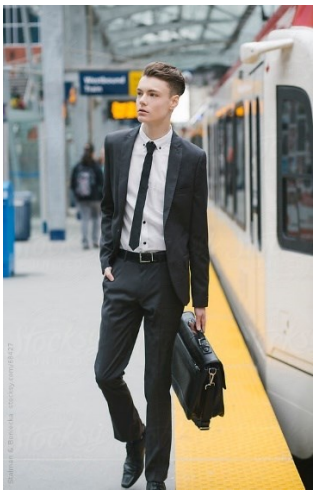
The example companies, organizations, products and domain names depicted herein are fictitious. No association with any real company, organization, and product or domain name is intended or should be inferred.

Disclaimer

Brian Leach Consulting Limited makes no warranty of any kind with regard to the material contained in this guide, including but not limited to the implied warranties or merchantability and fitness for a particular purpose.

The information contained in this document is subject to change without notice.

Meet Matthew



The Python Samples are the first series to be released with the MultiValue Bookstore sample application. They cover a series of options for working with the popular Python programming language within the context of the Rocket MultiValue platforms.

The samples revolve around Matthew, a school leaver who has been taken on by the bookstore as an apprentice under a government supported scheme. At school he took a computer class and learned Python and so has joined the existing IT team servicing the bookstore.

Matthew is keen to help and wants to learn about the MultiValue application they run and to make his mark.

This first series of samples illustrate three user stories, each of which presents a different way in which Python and MultiValue can work together. They are suitable for both MultiValue developers wishing to learn about Python, and for Python developers wishing to learn about MultiValue.

Starting Off

To run these samples, you must have installed the MultiValue Bookstore sample application on UniVerse for Windows. These samples will run with the free UniVerse Personal Edition.

Please see the MultiValue Bookstore - Getting Started guide for more information.

Python Sample 1

Sales Charting

Introduction

The Bookstore development team received the following user story into their backlog:

> **As a** Sales Director
> **I want to** visualize our sales performance across genres
> **So that** we can focus on new promotions.

'Hey, I can do that', said Matthew, 'That is easy under Python. There are loads of charting libraries and most of them work with pandas'.

The team looked at him strangely. 'What's this got to do with pandas?' asked Sarah whose desk had pictures of black and white mammals.

'Not those pandas', Matthew explained. 'It's a standard Python library for manipulating data using in data science. You load it with data and then lots of other packages use it. That's the way the Python community works, we all help each other.'

'Wow', said James, 'we could do with some of that here'.

The others ignored him

'So what would you need?' asked Sarah more helpfully.

'I just need a way to grab the sales figures out of UniVerse', Matthew answered.

'Well that's easy on our side', said James. 'UniVerse has two enquiry languages: a native English- like one that is really friendly, and a version of SQL that is more powerful. You can use either of those to query the data. I can show you how to write a statement to retrieve that data.'

'How do I get the results? Pandas can run queries against SQL databases but it's kind of slow and needs setting up.'

James thought for a moment. 'Universe can produce XML from any query - Is that any good to you?'

Matthew nodded, happily. 'Sure, I can create an element tree and just iterate through that. Sounds like I can bring in UniVerse data really easily. Then I can give you the chart as an image.'

'Great', said James. 'Can we pair on this? I want to see how you do it!'

Making it Happen

1. Proof of Concept

Matthew and James decided to pair on a simple stand-alone example first so that James could show Matthew how to get data from their UniVerse system and James could also evaluate what Matthew was doing. This example would be a simple proof of concept using a small file (the U2_BOOKS) just to plot the stock levels so they could iron out the technique quickly.

First, they went to get coffee.

Over coffee they decided to leverage the XML capabilities of UniVerse to create the results they needed, and then to parse that into a format that Matthew could use to produce his chart. Both of the UniVerse enquiry languages, RetrieVe and SQL, can be directed to generate results in XML format using a simple TOXML keyword, so this seemed the gentlest introduction.

So, to kick things off, James told Matthew how to write a simple UniVerse SQL query to produce XML and wrote down the following statement.

```
SELECT GENRE, SUM(STOCK_LEVEL) FROM U2_BOOKS GROUP BY GENRE TOXML;
```

Meanwhile Matthew had been looking at the u2py library and noticed that you can run any UniVerse command using the syntax:

```
result = u2py.run( command, capture=True)
```

The capture option would allow him to run the SQL enquiry and grab the XML results into a variable. Matthew already knew about the ElementTree in Python which is a simple means of iterating through an XML document looking for specific tags, so this would give him an easy way of working with the UniVerse data produced.

Matthew then used that data to populate a pandas DataFrame and passed that into matplotlib to create a simple line chart. This chart was stored in a directory as an image in png format, suitable for rendering on a web page or other device.

Because this was just an initial proof, he stores the image locally in a file named ‘stock.png’.

How to Run the Sample

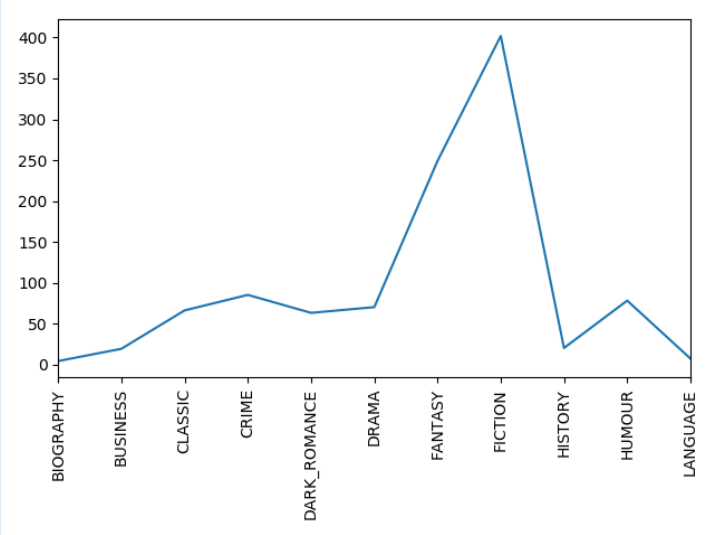
You can run a Python program from within the UniVerse command shell using the RUNPY command. This takes as its arguments the name of the file holding the script and the name of the script itself. The program runs directly inside the UniVerse process in the same manner as running a business language program using the RUN command.

To run the sample, you first need to start the UniVerse command shell: the easiest way is to navigate to the bookstore directory using your Windows File Explorer and double click the **start_uv.bat** file.

From there run the following command at the TCL (>) prompt:

> **RUNPY books.pysrc plotstockgenre.py**

This will create a new file named stock.png in the universe\mv_books directory.
You can open that using File Explorer.



HINT: If the commands you type change case this is due to a very annoying feature in UniVerse known as case inversion. You can switch this off using the command: PTERM CASE NOINVERT. You probably want to keep that one handy.

REMEMBER to type the OFF command to exit the UniVerse command shell when you have finished:

> **OFF**

2. Final Version

James was very pleased with the result - he wasn't sure about pandas but could see that it was easy to pass UniVerse data into Python. Matthew was also very pleased (and relieved) that he had delivered his first Python task successfully.

Now James could see how to produce a chart in Python using UniVerse data, but he knew that getting the figures for the sales charts would need to be a little more complex.

For a start, the data itself was more complex as the Bookstore holds sales orders in a MultiValued format using repeating detail lines for reasons of transactional efficiency. He would need to give Matthew a query that would undo the nesting to calculate the sales volumes:

```
SELECT BOOK_GENRE, SUM(QTY) FROM UNNEST U2_ORDERS ON BOOK_ID GROUP BY
BOOK_GENRE TOXML;
```

Then there was the fact that the chart should be more interactive:

- It would need to ask for a date range.
- It would need to tell the program where to store the image.
- It would need to create a new image every time.
- It would need to know what the new image was called.

Finally, they wanted to integrate this into their existing application that used UniVerse Business Language stored procedures known as subroutines. Could they call the Python code from a UniVerse subroutine?

James looked out the U2 Python manual and discovered that UniVerse can run stand-alone Python routines using the RUNPY command, and it can also call Python from within its business language, either as functions or by instantiating Python objects.

The PyCallFunction business language function runs a Python function in a named module:

```
Result = PyCallFunction(ModuleName, FunctionName, Params)
```

Using this, James could pass Matthew the details he needed to parameterize the query and set the location to write away the image. Matthew suggested he store each image under a GUID to ensure it was unique.

How to Run the Sample

You can run this sample from the UniVerse command shell by using the start_uv batch file as above.

The Python code is located in the books.pysrc u2_plotGenreSales.py script, but this is not designed to be called directly: the call is made from the books.bp u2_plotGenreSales Business Language subroutine. This sets the path for storing the image and converts the incoming dates for the query range into a standard format for the enquiry command.

You can see this being used from the web example (below) but for now you can also use a Business Language program that will prompt for those details and pass them into the subroutine.

Run the following command at the TCL (>) prompt:

> **U2.PLOT.GENRE.SALES**

HINT the bookstore sales data is between 2008 and 2009 on initial loading.

```
>U2.PLOT.GENRE.SALES

Start date : ?01 JAN 2008

End date : ?01 FEB 2008
Wrote plot at 9eca26a8-751d-44bf-9f64-887035568cda.png
```

This will write to the web\work directory under the bookstore.
Note: Files in this directory are ignored by git.

Python Sample 2

Recommendation Engine

Introduction

The Bookstore development team received the following user story into their backlog:

> **As a** Sales Director
> **I want to** improve our recommendations
> **So that** we can realize more cross-selling opportunities.

"This sounds like one of those simple stories that could get really complex" observed Sarah at their daily stand-up. She had not had her coffee yet and was feeling particularly cranky. "There's a hundred different ways to do this and they will spend forever trying to make up their minds on this one. Who's going to do the work?".

"Okay" said James with a sigh. "I'll go speak to the sales director and see if she knows what she wants."

"That would be a first", answered Sarah. "Why don't you prepare some different models to show her before we all get dragged in."

After speaking with the director, James informed the team they had decided to adopt a collaborative filtering approach similar to that used by other leading resellers (people who bought 'x' also bought 'y'). But it would also take account of what the customer had actually ordered in the past, so it would not offer them books they had bought previously.

This would need to have the following features:

- A recommendation would be based on a publication and optionally a client.
- The recommendations would look at who had bought that publication, and what other publications those same customers had purchased.
- Repeat orders would not be counted.
- The recommendations would initially be ranked on popularity based on numbers. It is assumed (naively) that all publications have been available for the same length of time.
- Weightings would then be applied as follows:
    - For the same author a 3x weighting would be applied.
    - For the same genre a 2x weighting would be applied.
- Finally, if a client was specified for the recommendation, any books they had already purchased would be filtered from the recommendation.

Making it Happen

1. Proof of Concept

"This still sounds like we're making too many assumptions", grumbled Sarah. "We need to do a simple proof of concept first to show to the business."

"Well", said James with an evil grin. "That sounds like you just volunteered".

To understand the requirements, a first attempt was made to build this using regular MultiValue techniques and the business language, by means of a Proof of Concept.

Jane wrote this using a standard technique for test driven development in UniVerse, creating a business language subroutine (books.bp u2_recommender_v1) that could be called with an Action code to select an operation, and with incoming and outgoing parameters that could be set and examined.

The Build action generates the initial recommendations database as a series of pairing records by running through all the orders in the U2_ORDERS file and writing combinations of titles purchased together to a new file named U2_PAIRS. The routine was written such that new orders could be added dynamically to the database when they get saved.

At the same time a new U2_CLIENT_BOOKS file also kept track of which clients had purchased which titles, to speed up the lookups and to filter out purchases by the client to whom the recommendations were being shown. This could have been fulfilled using the existing U2_CLIENT_ORDERS index but that would have slowed the retrieval of recommendations.

A Recommend action queries this database for a combination of a book number, client, and the option to apply weightings.

The solution proved workable in providing meaningful recommendations and was presented to the business to gain their acceptance on the decisions the team had taken. For once the business agreed: but pointed out that the routine was slow in building and the team decided it could be improved upon.

The Recommendation Algorithm

The recommendations are processed as follows:

```
For each book
        Get the full list of clients that have bought this book
        For each client
                Get the full list of books that the client has bought
                For each of the client books
                        Pair the book with the client book
                        Increment the count on that pair
```

How to Run the Sample

You can run this sample from the UniVerse command shell by using the start_uv batch file as above.

From the TCL (>) prompt type the command:

U2.RECOMMENDER.1

This will prompt you to first build the recommendations database if it has not previously been run. This may take several minutes to complete.

Once the database is there, you can search for a book by keyword (part of the author name or title) or by book number and it will serve you a list of recommended titles to go alongside it.

The build selects all the orders in the BOOK_SALES and runs through the purchased books adding each order to the recommendations (label BuildOrderToModel). For each book it updates a pair record that stores all other books that have been purchased alongside it and the number of such purchases. These are held as two multivalued fields with associated values forming a sub-table.

Once the database has been created it can be queried. The U2.RECOMMENDER.1 offers a simple keyword search on the title and author name as well as the title. The RECOMMEND action (subroutine label Recommend) reads the pairs and applies optional weightings to the numbers based on the author and genre before returning the top entries from the list. If a client id is given, those books previously purchased are first stripped from the list.

2. First iteration

After meetings with the team to review the Proof of Concept, it was decided that the design of the U2_PAIRS and U2_CLIENT_BOOKS were potential bottlenecks to the process. Whilst the recommendations were correct, it would be better to cache these details in memory following the initial build to improve performance.

Listening in on the meeting was Darren, a contractor who was finishing off some BI work for the business. He mentioned that he had previously used UDOs, the U2 Dynamic Objects supported by the Business Language. These would provide a good model for this, as these flexible JSON-style structures are ably suited to holding the key/value pairs that can match books with clients and vice versa.

Darren had some spare time whilst waiting for feedback from various departments, so he paired with Sarah and showed her how to use UDOs. The proof of concept was rewritten to store details using UDOs in place of new data files.

U2 Dynamic Objects (UDOs) offer a fast means of modelling information. Most combination variables in MultiValue are either static or dynamic arrays, designed to mirror the structure of the records or to collect groups of these, but UDOs offer more flexibility along the lines of JSON or Python objects. They can also hold large amounts of data in memory.

UDOs are created and manipulated through a library of Business Language functions. The sample uses the following functions:

UDOCreate - creates a new object or array.
UDOGetProperty - gets a named property along with its value.
UDOSetProperty - sets or creates a named property along with its value.
UDOArrayGetSize - gets the size of an array.
UDOArrayGetItem - gets an item from the array.
UDOArrayAppendItem - adds an item to the array.

This proved to be much more efficient, but Sarah in particular did not like the syntax of working with UDOs and expressed concerns about the maintainability and extensibility of this approach.

How to Run the Sample

You can run this sample from the UniVerse command shell by using the start_uv batch file as above.

From the TCL (>) prompt type the command:

U2.RECOMMENDER.2

This will prompt you to first build the recommendations database as the first example, but this should complete in a matter of seconds. Following this you can query the database in the same manner as the first recommender (and should get the same results!).

The BUILD action for the second recommender (label Build) is the same as that of the first, except that the files used in the first sample are replaced with two arrays: one storing Books to Clients and the other Clients to Books. Because these are fast to query there is no need to hold the pairs also. The lists are persisted in an area of memory known as a named common block: this retains them for the session, but the lists must be rebuilt every time you start a new session.

The RECOMMEND action runs first through the array of clients who have bought the specified book, and for each of those reads the other books they have bought, generating the pairing information dynamically. As before, these may have weightings applied and the client making the request is filtered from the list.

3. Final iteration

Matthew had returned to the team and they welcomed him back at their stand-up before discussing the status of the recommendation engine.

At this point, the team felt that they understood the problem domain and that the approach taken by the recommendation engine was correct. However, it would be better to develop this in a language that made the structure of the data more apparent and that would be easier to extend in future.

"You know, these UDOs sound just the same as Python variables", Matthew offered. As he explained, the team realized that Python has the same data structures as UDOs but would be cleaner and simpler to maintain. Moreover, because Python can run inside UniVerse it also has access to the same orders, books and client files as the business language without the need to run XML queries as they had for the charts.

There was one more issue that the team wanted to address. The UDO based recommender was a subroutine that could be bound into the application, but that meant every session would have its own copy of the entire recommendations array and so the idea of feeding new orders into it would not work as Sarah had imagined. What the needed was a way to have a single routine running constantly that would cache the recommendations and respond to requests from outside.

Matthew suggested writing the Python routine as a service that could run in the background and that would be fed new sales orders as they were placed. The team liked this approach, and James pointed out that the routine could run using the RUNPY command, as a UniVerse phantom process. All that was needed was a means for the two to communicate.

Matthew pointed out that Python has a fully serviceable simple http server. UniVerse and UniData also have built-in HTTP support, so the team were happy as they could call this from the Business Language using simple callHTTP requests.

You can also see this called directly from the sample web server (below).

How to Run the Sample

You can run this sample from the UniVerse command shell by using the start_uv batch file as above.

From the TCL (>) prompt type the command:

U2.RECOMMENDER.3

This is more involved than the previous sample, as you first need to start the recommendations engine as a background process (a phantom). This starts a new UniVerse process that runs the Python program using the RUNPY command you encountered in the first sample.

RUNPY books.pysrc u2_recommender_v3.py

The Python program listens for requests over an http connection. The requests are generated from a business language subroutine (books.bp u2_recommender_v3) using the callHTTP functions createRequest and submitRequest to send an http message and return the response.

Internally, the Python program builds the recommendations in the same fashion as the UDO based recommender by first directly opening and reading from the U2_ORDERS and U2_BOOKS files. The build method selects the orders file using the u2py.List() function, which produces an iterable series of keys to the orders. These are then used to read the associated order and pass it into the buildOrderToModel method.

The buildOrderToModel converts the order record into a Python list for ease of access: each field in the order is turned into a numbered list element starting from zero. The MultiValued books on the order are turned into a sub-array.

The method examines the books on the order and adds them to the two arrays. The Python bisect library is used to quickly search the arrays.

NOTE:

You may have to kill the web server using Task Manager.

Python Sample 3

Web

Introduction

Following the success of their first Python examples, the team were excited to learn more and to demonstrate their new Python skills. They realized that their existing Windows front end was not really showing this in its best light.

So the team asked the head of development to raise a user story on their behalf:

> **As a** Development Manager
> **I want to** create some web pages
> **So that** we can demonstrate our application.

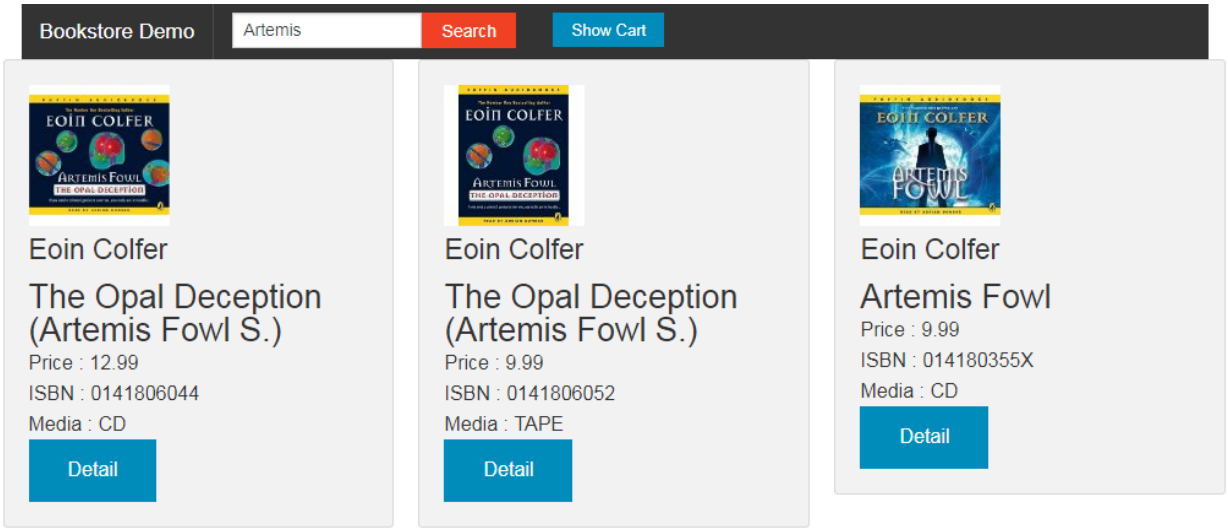
Making it Happen

The new recommender has shown the team how easy it was to create an http listener using Python, and how well that could talk to UniVerse. Surely, they reasoned, this could be used to service more than the recommender? Could it be used to serve web pages and other web API requests?

This time, Matthew paired with Sarah to build a framework web server that could return pages using code found on the internet. They discovered the code was for Python 2, but in converting it to Python 3 Sarah learned a lot about how to work in a Pythonic manner.

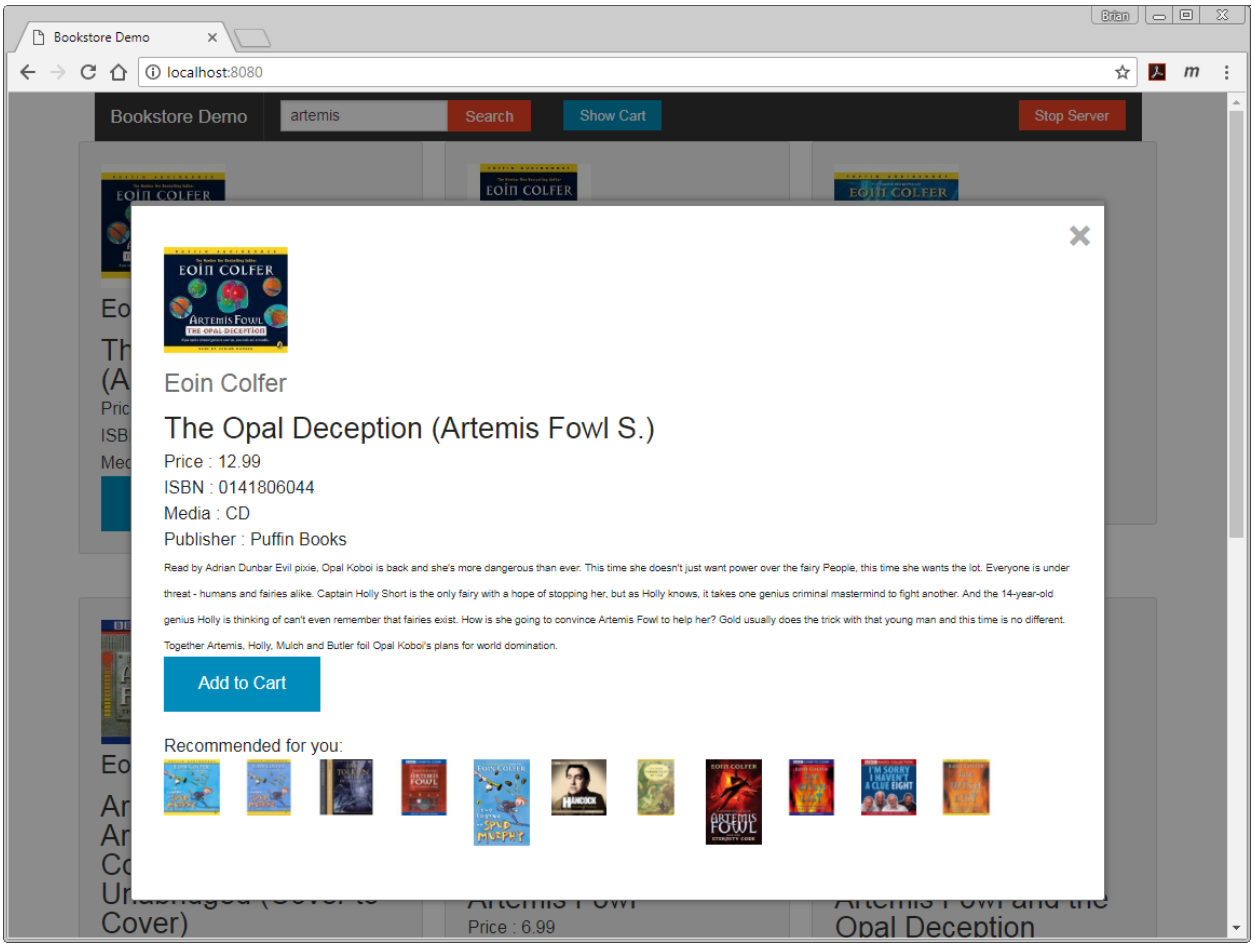
Meanwhile, James worked with rest of the team to add in a set of methods that would interact with their UniVerse application. They decided on a convention of /api/methodname to call the method, and Matthew suggested that each method should be a separate class for ease of maintenance.

Before long they had a working web server that could model their bookstore as a Single Page App:



The team liked the new web server (though they argued long about the look and feel of the pages!).

But they did not like the fact that they now had two HTTP listeners to maintain. Could they be combined into one? The team added the recommendation engine directly into the web server.



How to Run the Sample

So far, all the Python code has been run from within a UniVerse session, either using the RUNPY command or calling the code from a business language routine. But it is possible to run Python routines from outside an existing UniVerse session.

Unlike those other samples, this runs from a regular Python session but must be invoked using the version of Python installed with UniVerse.

The start_webserver.bat file that is installed in the mv_books account folder runs the books.pysrc webserver.py script using the correct version of Python.

The webserver.py acts as a simplified web server offering pages from under the bookstore web directory (where the charts were written in the first sample). The webserver reads the http request for a URI will send back regular pages and images. The web pages are written in equally simple html with some jQuery callbacks to fetch data from the webserver.

However, if the URI begins with a special path the webserver will branch to some additional processing.

For URIs starting with ‘re’, it will call the recommendation engine. This is imported into the webserver by sharing the u2_recommender_v3.py script above.

For URIs starting with ‘api’, it looks into a separate module of functions named bookstore_modules.py, also in the books.pysrc file. These contain simple functions for selecting books, reading a book detail and adding a cart to the U2_ORDERS file.

The cart processing has not been finished: it only asks for the client id. There is an opportunity for you to test your Python knowledge and see if you can assist the team!