Rapid Application Development

GAE Templates and Multiple Pages

Coventry University

Outcomes

Understand and apply the MVC pattern

Understand the benefits of templates

Build web apps using the Jinja2 template library

Build and link multiple web pages

Host static files

Quick Recap

```
class MainHandler(webapp2.RequestHandler):
    def get(self, bookid):
      lang = self.request.get('lang')
      if len(lang) > 0:
        self.response.write('<h1>Preferred lang: '+lang+'</h1>')
      link = '<p><img src="http://'+self.request.host+'/thumbnails/'+bookid+'.jpg" /></p>'
      self.response.write(link)
```

Templates

While we could write all of the HTML into the response using self.response.out.write(), we really prefer not to do this

Templates allow us to separately edit HTML files and leave little areas in those files where data from Python gets dropped in

Then when we want to display a view, we process the template to produce the HTTP Response

How Templates Work

A well written App Engine Application has no HTML in the Python code:

- it processes the input data, talks to databases, makes lots of decisions, figures out what to do next and then
- Gets some HTML from a template - replacing a few selected values in the HTML from computed data

What is a Template?

An HTML file sent to the browser

Contains placeholders which are replaced by data from the server

The Python script loads the template and passes it the data

No HTML code in the script

Results in a script that is easier to understand (and edit)

Separation of Concerns

The Python handler script contains the code to:

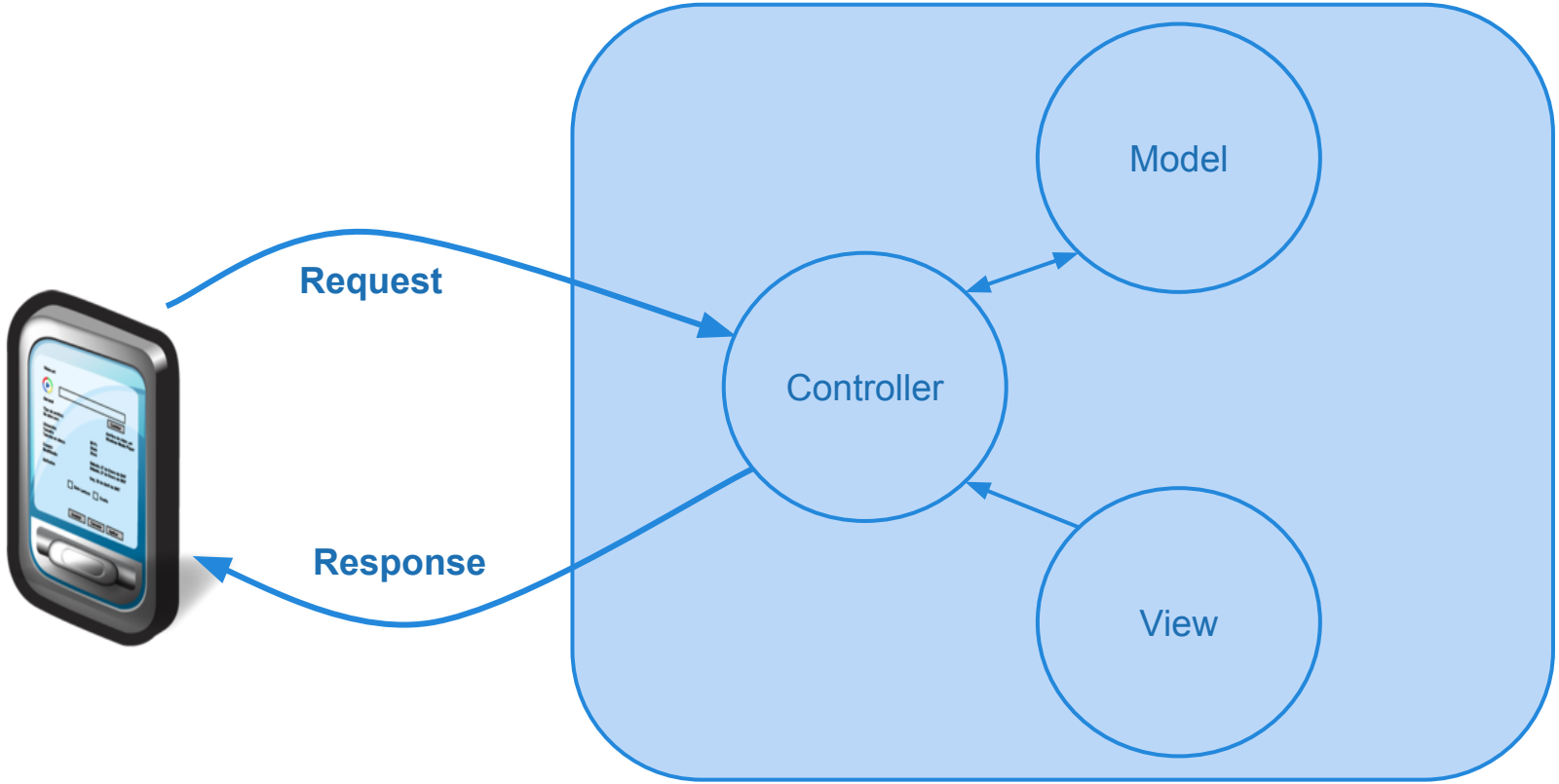
- retrieve data
- perform calculations and logic
- detect user interaction

The Template file:

controls how the page will be rendered

Keeping this separation we build code that is maintainable

Model-View-Controller



Typical Template Structure

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title></title>
</head>
<body>
    <h1>Preferred Language: {{lang}}</h1>
    <table>
        {% for book in books %}
        <tr>
            <td><img src="/thumbnails/{{book.id}}.jpg"/></td>
            <td>{{book.title}}</td>
        </tr>
        {% endfor %}
    </table>
    <p><img src=""/></p>
</body>
</html>
```

Google App Engine and MVC

HTML in the templates is an example of a **View**

The various handlers are example of **Controller**.

The MVC enable us to organise the functionality of our application well and to deal with complexity as we add more functions and features.

Very important to acquire good habits as we learn in order to become good developers.

Model-View-Controller

HTML in the templates is an example of a View

The various handlers are example of Controller.

The MVC enable us to organise the functionality of our application well and to deal with complexity as we add more functions and features.

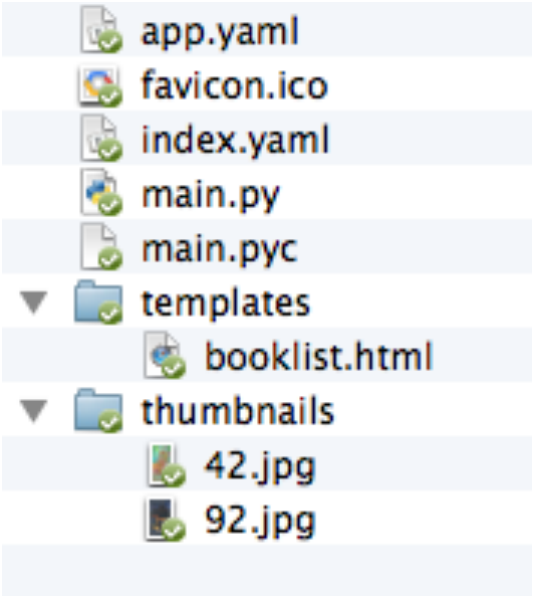
Very important to acquire good habits as we learn in order to become good developers.

Folder Structure

The HTML templates should be kept in their own folder

This example uses a folder called templates

The name is not important!



The Jinja2 Templating Language

About Jinja2

Jinja2 is a templating engine for Python.

Useful Links

The Jinja2 Website
Jinja2 @ PyPI
Jinja2 @ github
Issue Tracker

Versions

Development (unstable)

Quick search

Welcome to Jinja2

Jinja2 is a modern and designer-friendly templating language for Python, modelled after Django's templates. It is fast, widely used and secure with the optional sandboxed template execution environment:

```
<title>{% block title %}{% endblock %}</title>
<ul>
{% for user in users %}
  <li><a href="{{ user.url }}">{{ user.username }}</a></li>
{% endfor %}
</ul>
```

Features:

- sandboxed execution
- powerful automatic HTML escaping system for XSS prevention
- template inheritance
- compiles down to the optimal python code just in time
- optional ahead-of-time template compilation
- easy to debug. Line numbers of exceptions directly point to the correct line in the template.
- configurable syntax

Application Configuration File

If we want to use the jinja2 framework

We need to import it in our config file

```
libraries:
- name: webapp2
  version: "2.5.2"
- name: jinja2
  version: "2.6"
```

The Python Script

Needs to include the following:

- import the correct frameworks
- create the JINJA_ENVIRONMENT variable

In the get method:

- create a template variable
- assemble the data
- render the template passing the data

```
#!/usr/bin/env python

import jinja2
import webapp2
import os

JINJA_ENVIRONMENT = jinja2.Environment(
    loader=jinja2.FileSystemLoader(os.path.join(os.path.dirname(__file__),'templates')),
    extensions=['jinja2.ext.autoescape'])


class MainHandler(webapp2.RequestHandler):
    def get(self):
        template = JINJA_ENVIRONMENT.get_template('booklist.html')
        myBooks = [{'id':42, 'title':'THGTTG'}, {'id':92, 'title':'Foundation'}]
        template_values = {
            'lang': 'en',
            'books': myBooks
        }
        self.response.write(template.render(template_values))

app = webapp2.WSGIApplication([
    ('/', MainHandler)
], debug=True)
```

Import Frameworks

Need to import the os and jinja2 templates

Environment Variable

use the path to the templates and create the variable for later use

Load the Template

From the environment variable we created earlier

Building the Data

Create a set of key-value pairs

Render

Render the template and write to the response object

Libraries

the JINJA_ENVIRONMENT global variable using the statement

```
JINJA_ENVIRONMENT  = jinja2.Environment(loader=jinja2.
FileSystemLoader(os.path.join(os.getcwd(),’templates’)))
```

Building and Linking Multiple Pages

Challenge

Most websites have multiple pages

We want to be able to add more pages to our application

Options

REMEMBER: each page needs a controller (Python class)

There are two options:

- Create multiple controllers in the same file for different pages
- Create a separate file for each controller


SINGLE FILE
MULTIPLE FILES

The Best Choice

In simple terms:

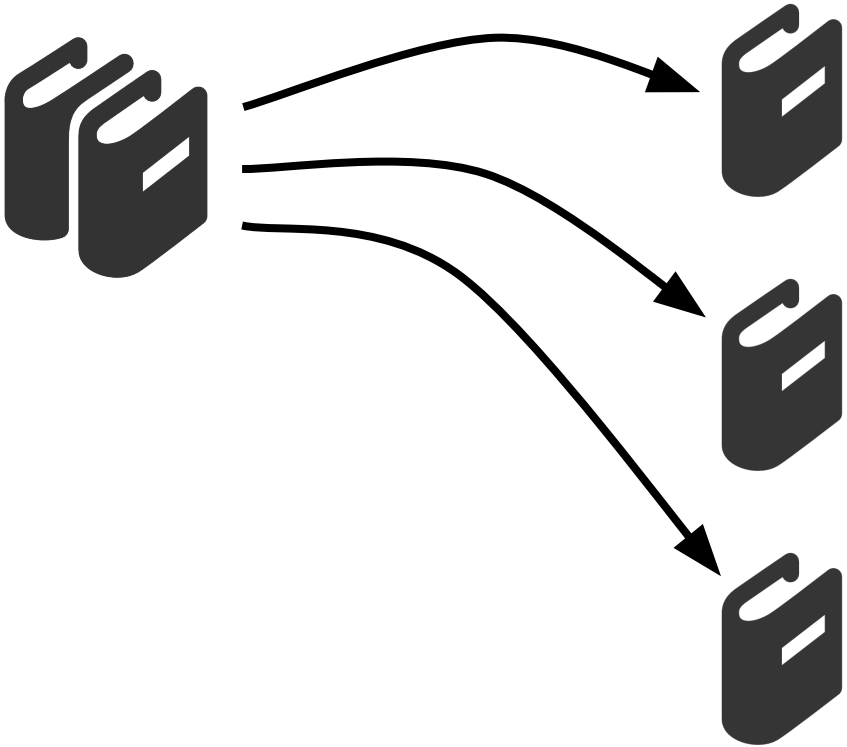
Whatever makes more sense to you the developer…

My preference is for separate controller files for each page

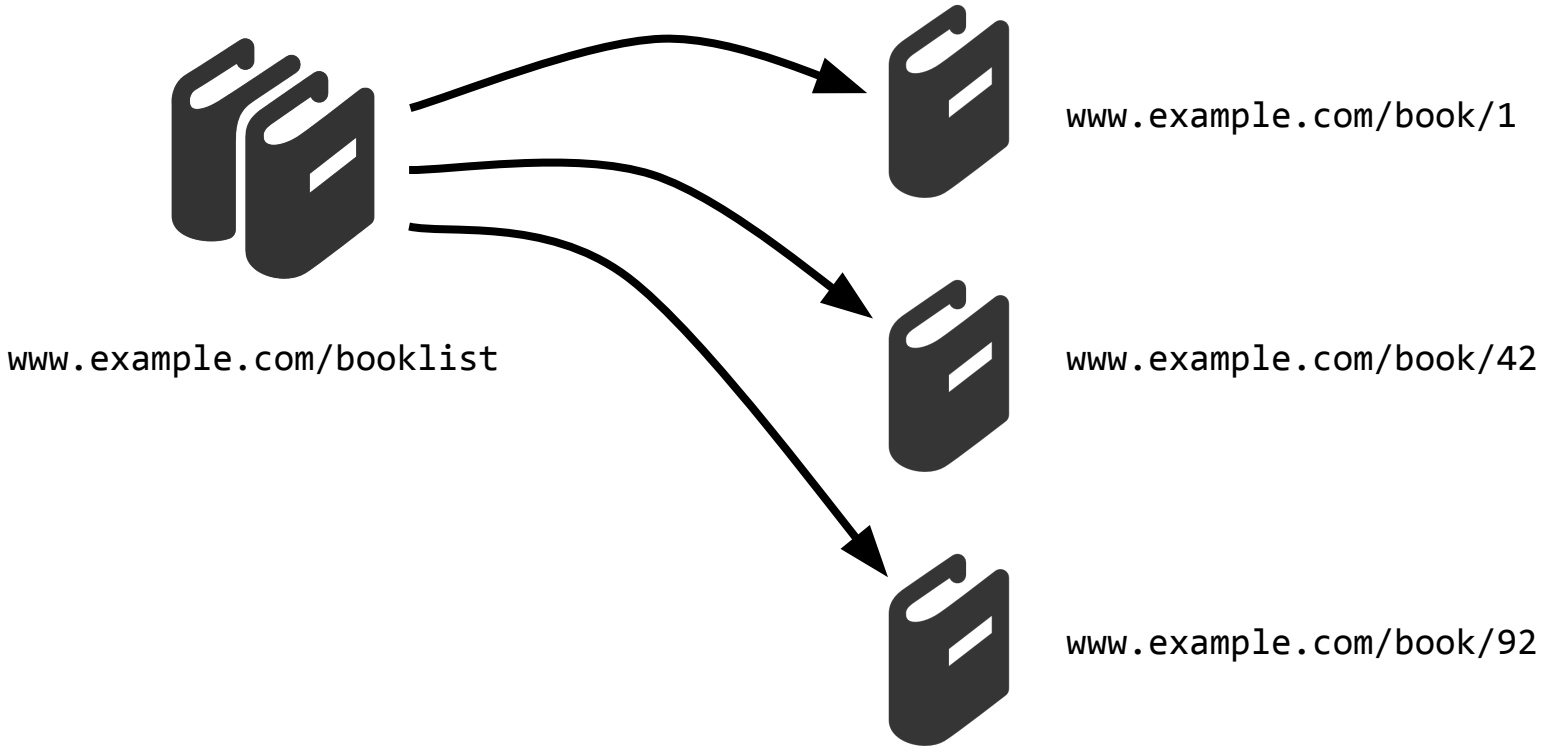
Keeps each file short and easy to follow

Extending the Application

URL Structure



URL Structure



Challenge

Consider the URL:

```
http://localhost:8080/book/42?lang=en
```

The challenge is to display not only the book id and lang preferences in the browser but also display the correct book cover

Outcomes

To enable our app to understand this we need the following:

- A directory containing the book covers
- A new handler to handle different URL paths (book)
- To create a new python script to handle book requests
- To be able to read URL segments (42)
- To be able to read URL parameters (lang=en)
- To be able to reference files in our static files directory

Adding a Directory for Static Files

Resources are dynamically generated

To serve static files we need to create a directory

Then add a handler to pass the requests to this

See next slide for details

Adding a new Handler

Need to create a handler to route specific URL to the directory

Can create a route to a specific file pattern

Or to a whole directory

Specify a file pattern using the static_files handler

Specify a directory using the static_dir handler

Handlers listed most specific to least specific

```
handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: /thumbnails
  static_dir: thumbnails

- url: .*
  script: main.app
```

static_files handler

A pattern representing specific files on the server

url

The URL path we want to map to the directory

static_dir handler

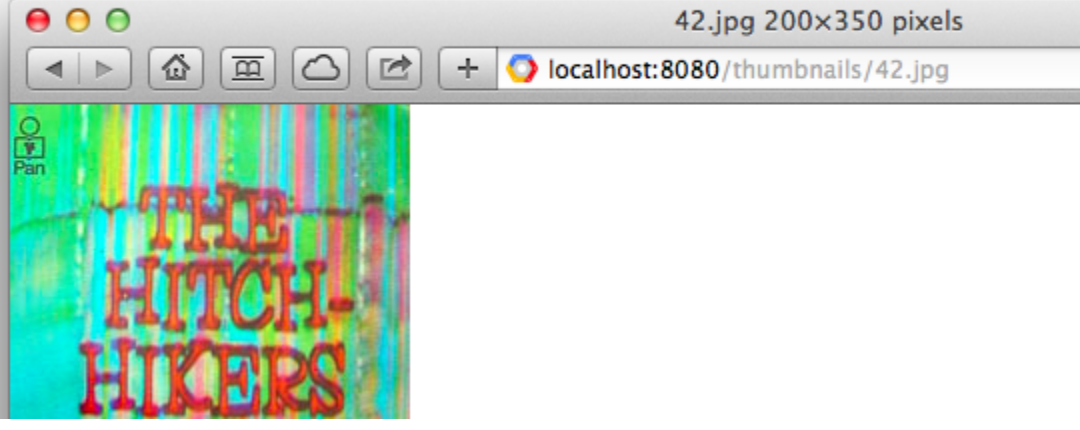
The name of the physical directory on the server

Viewing Static Files

URL contains the correct route and filename

Static files can be accessed in the browser

```
http://localhost:8080/thumbnails/42.jpg
```



Adding a Second Script

If the URL includes a segment called books

We want to use a different python script

There are two steps:

Create a second python script

Add a handler in the app config file

```
#!/usr/bin/env python

import webapp2

class MainHandler(webapp2.RequestHandler):
    def get(self):
      self.response.write('Hello Book!')

app = webapp2.WSGIApplication([
    ('/book.*', MainHandler)
], debug=True)
```

Response

Lets display something unique so we can see if it is working!

Path

matches /book followed by 0 or more characters

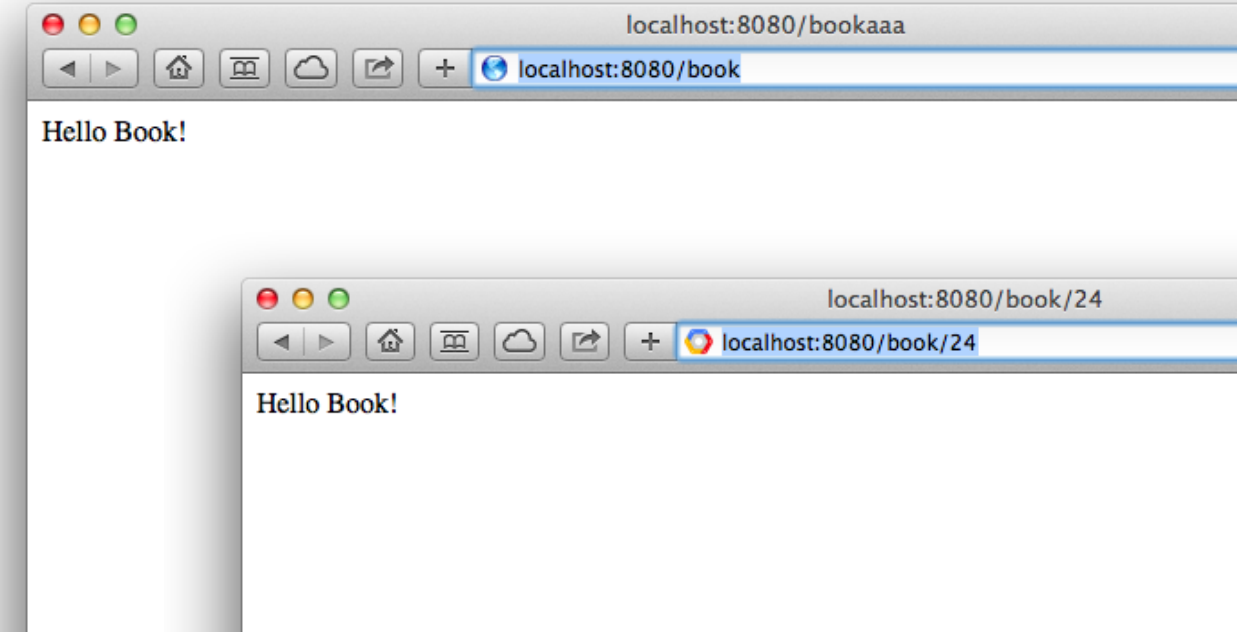
```
handlers:
- url: /favicon\.ico
  static_files: favicon.ico
  upload: favicon\.ico

- url: /thumbnails
  static_dir: thumbnails

- url: /book.*
  script: books.app

- url: .*
  script: main.app
```

Handler

This handler matches /book folloed by 0 or more characters

Testing The Handler and Script

Note how the handler can handle additional characters



Reading URL Segments

Consider the URL:

```
http://localhost:8080/book/42
```

The assumption is that by passing a different value we would access different books

For this to work we need to extract the value from the URL

This can be achieved using a regular expression

```
#!/usr/bin/env python

import webapp2

class MainHandler(webapp2.RequestHandler):
    def get(self, bookid):
      self.response.write('Book ID:'+bookid)

app = webapp2.WSGIApplication([
    (r'/book/(.*)', MainHandler)
], debug=True)
```

Additional Parameter

The regular expression evaluated inside the brackets passed as a parameter

Regular Expression

Anything in brackets gets stored and passed to the method in MainHandler

URL Parameters

URL parameters appear at the end of the main URL and take the form of key-value pairs

For example:

```
http://localhost:8080/book/42?lang=en&sort=desc
```

They are typically used to modify data returned rather than specify the data itself.

For example changing the language used or the sort order for tabular data

Reading URL Parameters

All URL parameters are accessed through the get property of the request object

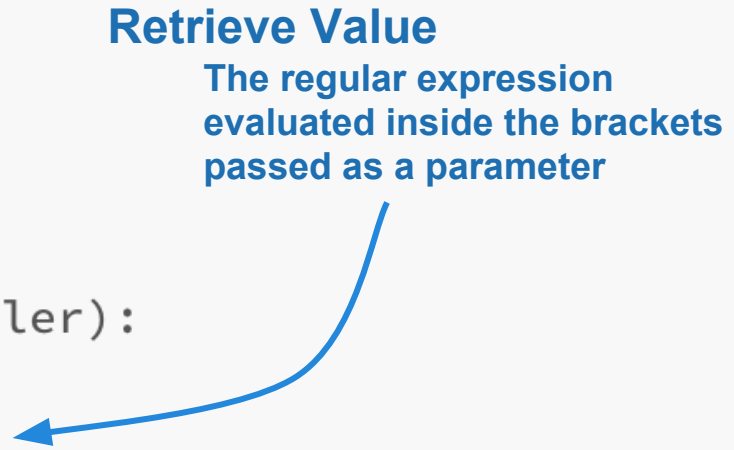
Specify the key

Returns the value

```
lang = self.request.get('lang')
```

If the key does not exist it returns an empty string

```
#!/usr/bin/env python

import webapp2

class MainHandler(webapp2.RequestHandler):
    def get(self):
      lang = self.request.get('lang')
      self.response.write('Preferred lang: '+lang)

app = webapp2.WSGIApplication([
    ('/book.*', MainHandler)
], debug=True)
```



Checking That Parameter Exists

If the parameter doesn’t exist

request.get returns an empty string

We can check this using a simple conditional

```
#!/usr/bin/env python

import webapp2

class MainHandler(webapp2.RequestHandler):
    def get(self, bookid):
      lang = self.request.get('lang')
      if len(lang) > 0:
        self.response.write('<h1>Preferred lang: '+lang+'</h1>')
      link = '<p><img src="http://'+self.request.host+'/thumbnails/'+bookid+'.jpg" /></p>'
      self.response.write(link)

app = webapp2.WSGIApplication([
    (r'/book/(.*)', MainHandler)
], debug=True)
```

Python Conditional

Check that the parameter exists before using it.

The Completed Application

You should now have enough knowledge to be able to build simple multi-page applications

Adding a New Page

Create a new Python controller file

Add a new route to the app.yaml file

Lets see an example...

Summary

You should now be able to:

- Understand and apply the MVC pattern
- Understand the benefits of templates
- Build web apps using the Jinja2 template library
- Build and link multiple web pages
- Host static files