Capstone Retail Black Friday Project – Internal Architecture

Table Of Contents

1 Capstone Web Application

The Capstone base contains the same sample web application in both python and Java.

- Some sample web pages using the Jinja web templates
- A sample web template that contains a google chart
- A rest API with 2 apis that can retrieve Cassandra data and return it in a format which works for google chart.
- A very light MVC Web framework

Internals of each one:

The applications are divided into the following components as diagrammed below. In addition, there is a description of how these components are implemented in Java and Python.

1.1 View - Jinja Web Templates

Both the Java and Python applications use jinja template. On Python, we are using Flask as our web framework, and Jinja2 is an integral part of it. Not wanting to rewrite the templates in a different language, the Java app includes jinjava to process Jinja2 templates. The templates have been carefully crafted as to keep them common between the 2 platforms. To do this we:

- Did not use any language-specific syntax
- Stuck to standard relative paths for included templates, and URLs eg. /base.jinja2
- Created and registered a single helper function in both languages called makeURL as opposed to adding any fancy language code.

For Java, we also treat the JSON output like a view, and have a method to convert a Cassandra ResultSet to json in the correct format.

Model - We skipped these in python, but implemented it in Java. The model objects represent entities in the data model, and the operations on it. An entity may involve one or more Cassandra tables.

Controllers - controllers handle the flow of the application. They receive a web request, call the model to retrieve or store data, and then return the results or the next web page. In the case of the rest api, they simply return the JSON result. For the regular web pages, they render the appropriate Jinja2 templates. The frameworks each have their own way of mapping URLs to the appropriate method or function to server that URL.

1.2 Some Helper Classes

Cassandra Object - This manages the connections and sessions to Cassandra. It implements a single connection, and it is assumed the nodes in that data center are adequate for both the Cassandra workload and Solr workload.

JinjaHelper - Functions that are used in the jinja templates.

jsonoutput - Functions to assist with building json for google charts. Not used in Python.

The Mainline - This configures the web application framework and starts it. It reads the application.cfg file and configures the port, the Cassandra connection, and Jinja helpers, etc.

2 Jinja Details (Java & Python)

Jinja2 is a simple templating language used in this this project for rending dynamic web content. To render a web page, simply call the rendering function, and pass in all of the parameters you reference in the template. There are examples in each platform of rendering the templates.

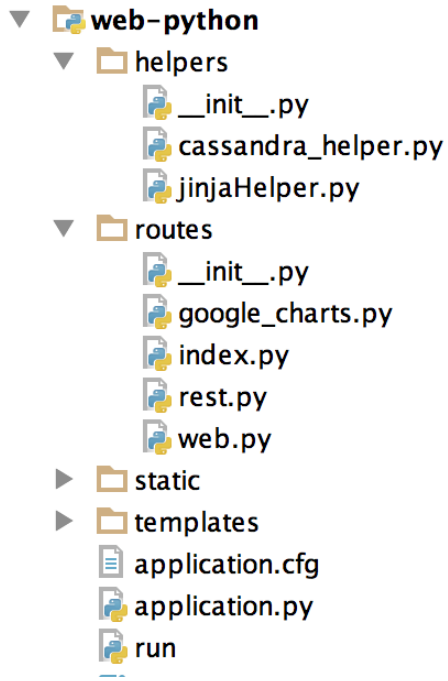
The project just uses a couple of simple features of the jinja templates.

- {% <some code> %} - we use some simple code that is supported in jinjava. For java, it's a bit restrictive, whereas for python the block may contain any python expressions. Some simple statements are for, set, if, and a method call

- {{ <variable expression }} - This simply outputs the given expression. In Java, you can dereference fields in beans so long as the getter is in CamelCase. If the bean as getProductName(), we can have expression in the template like {{ product.product_name }}

2.1 Python Implementation

2.1.1 Code Tree



Mainline
Application.py - This registers python objects to serve URL patterns, and registers the makeURL function to Jinja2.

Registering new routes (URL to python function):
Routes are registered in 2 phases:

If you are creating a new python file, register it in the file as follows: Creating a new Blueprint for the file:

```
web_api = Blueprint('web_api', __name__)
```

Then Give the file a URL pattern and register it in the mainline (Application.py)

```
app.register_blueprint(web_api, url_prefix='/web')
```

1) In the file you have created or are extending, create functions for next part of the URL. You may have parameterized parts of the URL pattern in the case you are implementing REST-style URLs. In the example below, <series> is a parmeter.

```
@rest_api.route('/realtime/<series>')
def timeslice(series=None):
```

Implementing the function:

If you have any query parameters on the URL, retrieve them with the request.args.get method. This example gets the minutes query. If the URL is <url>?minutes=....

```
request.args.get('minutes', 5)
```

If the query string is not present, it will default to 5. Then implement your logic to store and retrieve information from Cassandra, and then render and return the result. This example renders a template passing product as a parameter:

```
return render_template('product_list.jinja2', products = results)
```

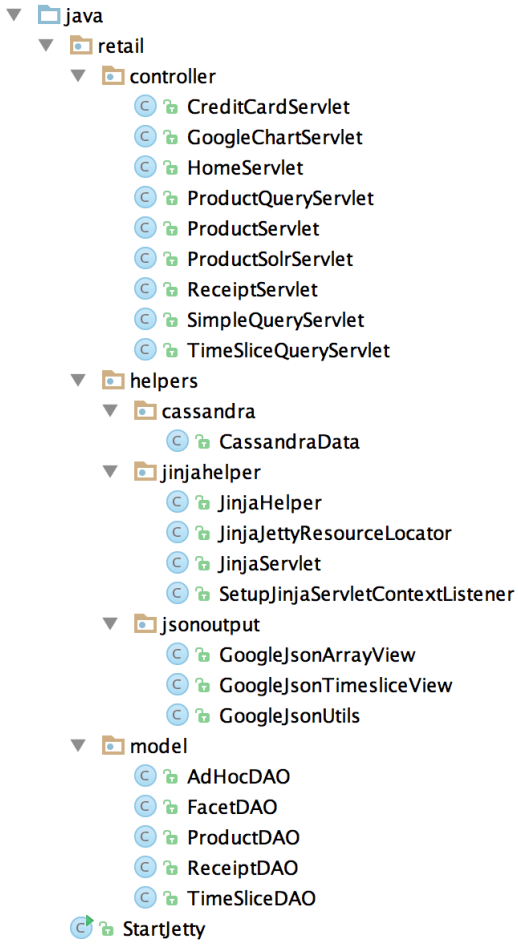
In the case of json, simply return the json string you build up:

```
thejson = dumps([description] + data, default=fix_json_format)
return thejson
```

The default=fix_json_format, call a method in rest.py to override the way certain datatypes are rendered. For example, dates are rendered as Date(m,d,y,h,m,s), a format used by google charts.

2.2 Java Implementation

2.2.1 Code Tree



2.2.2 Mainline

The mainline is StartJetty, and sets up the web application to handle the servlets and the static content. Note that the web.xml registers SetupJinjaServletContextListener

to set up some jinja2 things, and hang it off of the ServletContext so it is accessible to the servlets.

2.2.3 Registering Routes

The Java implementation is using embedded Jetty with pure servlets to implement methods to service URL requests. In the retail sample application, there is generally a single servlet for each web page, but you may design servlets which serve several pages each. To register a class as a servlet, edit the webapp/META-INF/web.xml file. You register both the servlet, as well and the URL pattern or patterns that go with it.

```
<servlet>
   <servlet-name>ProductServlet</servlet-name>
   <servlet-class>retail.controller.ProductServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>ProductServlet</servlet-name>
   <url-pattern>/web/product</url-pattern>
</servlet-mapping>
```

Implementing the servlet:

If your servlet is using the Jinja framework, it should extend JinjaServlet, otherwise it may extend HttpServelt. The Jinja servlet adds a render method which handles rendering the template with the correct encoding.

Retrieve the URL query parameters with the request

```
request.getParameter("product_id");
```

If you want a rest-style API, you need to parse out the URL string.

In the Java application, we have implemented DAO objects, so you should call them to interact with Cassandra. While a servlet can certainly make direct Cassandra calls, it runs counter to the MVC pattern.

```
ProductDAO product = ProductDAO.getProductById(product_id);
```

Then render the template. Place the template parameters in a Map<String,Object>, and call render. You need to write the rendered template to the servers output stream. This is necessary as the render function encodes the output in UTF-8 for browser compatibility.

```
Map<String, Object> context = Maps.newHashMap();
context.put("product", product);
byte[] renderedTemplate = render("/product_detail.jinja2", context);
out.write(renderedTemplate);
```

If the servlet return JSON
There are a number of helper functions in the jsonoutput to convert a ResultSet into a google JSON array. GoogleJSONArrayView works on regular rows. The special timeslice version expects each row to contain a map called quantities, and it will make each element of the map look like its own column.

2.3 Working With DAO Objects

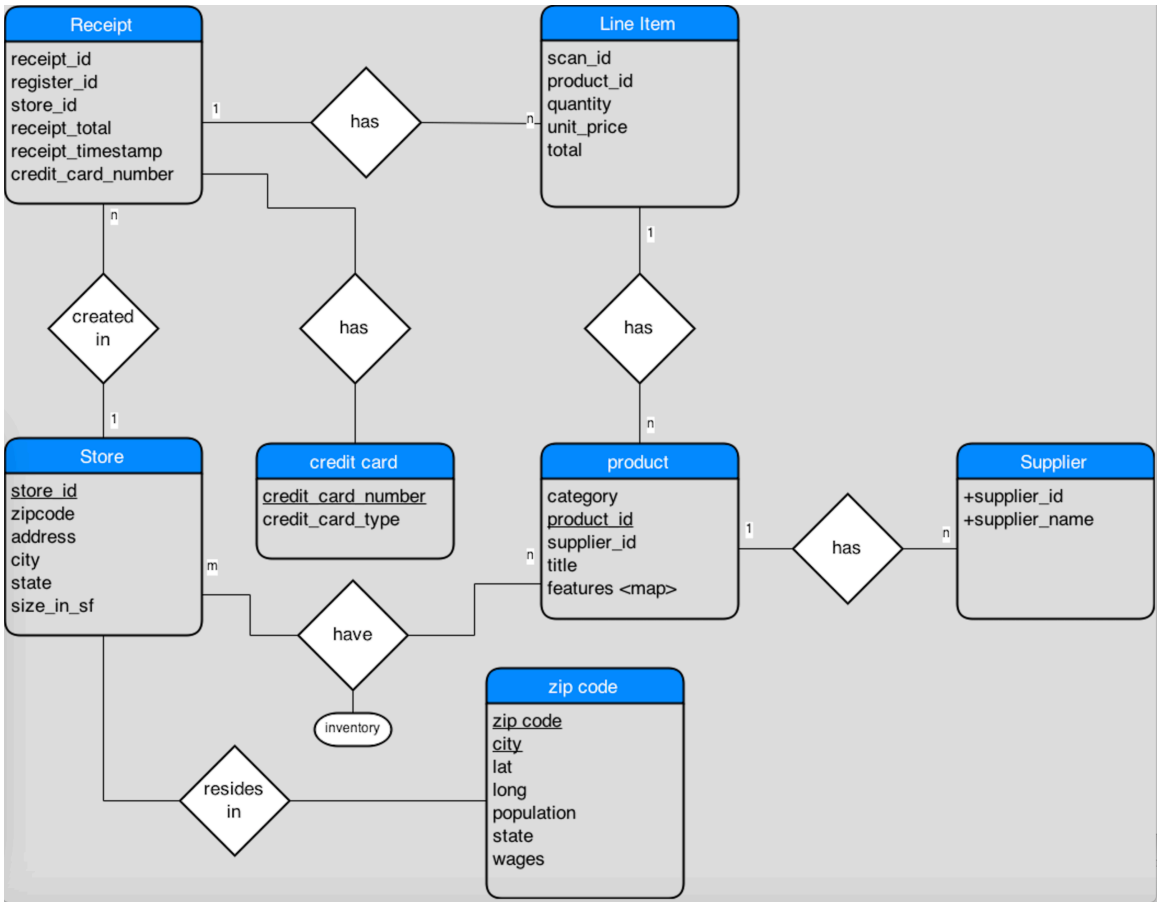
Our DAO Objects in the model package are simply Java Beans that can be constructed from a cassandra Row object. In addition it has several static method to query the database and return a list of DAO objects. The DAO objects in this framework generally extend CassandraData, so they can call the method loadBeanFromRow to save code in copying the data from the cassandra row to the bean fields.

Alternatively, you can write your own constructor like the code below. loadbeanfromrow loops through all of the fields in the current row, and tries to find the setter function for that field. The setter must be of the form setSomeField for it to work.

```
productId = row.getString("product_id");
categoryId = row.getInt("category_id");
```

The DAO static methods will return a List<someDAO> or simply a someDAO if it is a singleton select. The special DAO objects AdHocDAO and TimeSliceDAO simply return ResultSets

3 ER Datamodel


Receipt
receipt_id
register_id
store_id
receipt_total
receipt_timestamp
credit_card_number
has
Line Item
scan_id
product_id
quantity
unit_price
total
created in
has
has
Store
store_id
zipcode
address
city
state
size_in_sf
credit card
credit_card_number
credit_card_type
product
category
product_id
supplier_id
title
features <map>
has
Supplier
+supplier_id
+supplier_name
have
inventory
zip code
zip code
city
lat
long
population
state
wages
resides in
1
n
m