

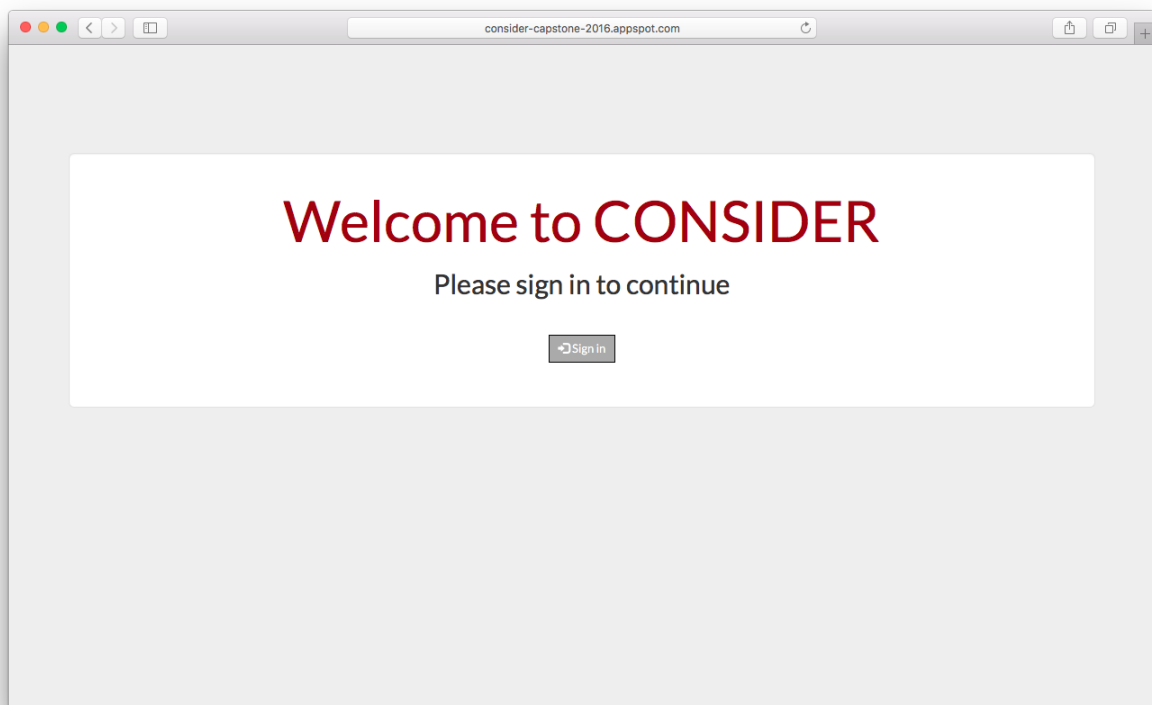
**CONSIDER**

# Architecture and Design

by **Andrew Clinton, Don Herre, Tyler Rasor, Dustin Stanley**

April 25, 2016

---



---

## Intro

This document will provide a technical overview of the underlying architecture for the CONSIDER project. By the end of this paper, the reader should have a firm understanding on how data is modeled and related, the MVC design, and how the app functions as a Web API.

## Table of Contents

<u>Framework</u>	2
<u>DataStore Schema</u>	2
<u>MVC Design</u>	3
<u>Models</u>	3
<u>Views</u>	3
<u>Controllers</u>	4



## Framework

This app was built on top of Google App Engine as it provides easy to configure servers which are managed by Google allowing developers to focus on developing powerful apps. GAE allows developers to choose from several languages & frameworks when building a project. CONSIDER is built upon webapp2 which is a python web framework. Webapp2 enables python developers to rapidly build functional prototypes using a NoSql datastore.

## DataStore Schema

GAE allows for several different types of storage types:

1. App Engine Datastore
2. Google Cloud SQL
3. Google Cloud Storage

CONSIDER uses the App Engine Datastore which is a NoSql variant. To represent the number of data entities in the app, there are several models:

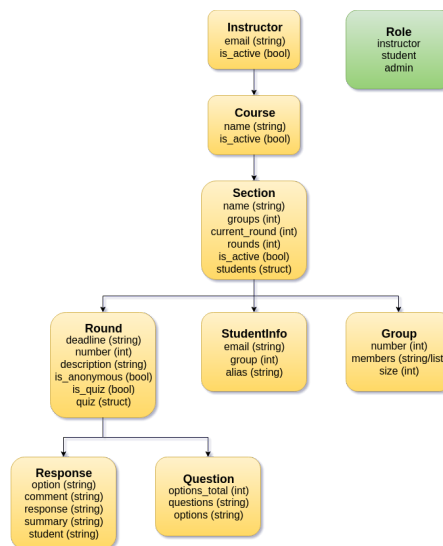


Figure 1

## MVC Design

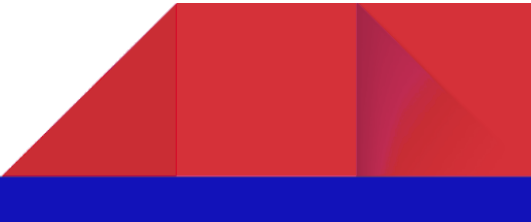
Webapp2 follows the Model - View - Controller design paradigm which allows for the proper separation of concerns.

### Models

The model structure for CONSIDER can be seen in Figure 1 above. The various models are laid out in such a way that it is easy to create parent - child relationships which is especially important for the Round model.

### Views

Webapp2 allows for several different types of template engines. CONSIDER uses the Jinja2 engine. By using Jinja2 developers have the ability to use powerful looping mechanisms and logical blocks. **Figure 2** below shows an example of a for loop and if/else logic blocks in one of the instructor views.



```
156     </thead>
157     <tbody>
158     {% if discussionRounds %}
159         {% for obj in discussionRounds %}
160             <tr id="round_{{obj.number}}" {% if selectedSectionObject.current_round == obj.number %}
161                 <td>
162                     <button data-section="{{selectedSection}}" data-course="{{selectedCourse}}" data-
163                         <span class="glyphicon glyphicon-remove"></span>
164                     </button>
165                 </td>
166                 <td>{{obj.number}}</td>
167                 <td class="round_starttime">{{obj.starttime|default('n/a')}} <input type="hidden" val
168                 <td class="round_deadline">{{obj.deadline}} <input type="hidden" value="{{obj.deadlin
169                 <td class="round_description md">{{obj.description}} <input type="hidden" value="{{ob
170                 <td>
171                     <button data-round-key="{{obj.number}}" class="btn btn-default edit-round" aria-l
172                         <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
173                     </button>
174                 </td>
175             </tr>
176         {% endfor %}
177     {% endif %}
178     </tbody>
179 </table>
180 </div>
```

Figure 2

## Controllers

Webapp2 routing allows for the standard HTTP verbage:

1. GET
2. POST
3. PUT
4. DELETE

In order to capture requests for the different types of verbs, a developer can create routes to match urls to specific python classes which handle the action verb sent from the user. **Figure 3** shows an example of this routing mechanism.

Figure 3

```
102 application = webapp2.WSGIApplication([
103     ('/', MainPage),
104     ('/home', MainPage),
105     ('/error', ErrorPage),
106     ('/admin', admin.AdminPage),
107     ('/courses', Instructor.courses.Courses),
108     ('/sections', Instructor.sections.Sections),
109     ('/students', Instructor.students.Students),
110     ('/rounds', Instructor.rounds_test.RoundsTest),
111     ('/test-rounds', Instructor.rounds_test.RoundsTest),
112     ('/responses', Instructor.responses.Responses),
113     ('/group_responses', Instructor.group_responses.GroupResponses),
114     ('/groups', Instructor.groups.Groups),
115     ('/student_home', student.HomePage),
116     ('/student_rounds', student.Rounds),
117 ], debug=True)
118
119
```

Each python controller class can have 4 methods which will respond to each of the 4 different HTTP verbs. **Figure 4** shows an example of a controller handling Instructor related logic.

Figure 4

```
37 class RoundsTest(webapp2.RequestHandler):
38     def get(self):
104         #end get
105
106     def post(self):
157         #end post
158
```