1 Homework 3 Solution

Due Date: Nov 14; To be turned in on paper in class.

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

Please remember to justify all answers.

You are encouraged to use the web or the library but are required to cite any external sources used in your answers.

1. The mustache template given in the recursive.js example discussed in class depends on one of the bad parts of JavaScript and would not work in other languages like Ruby. Identify why that is so and how would you fix it to remove the problem? *15-points*

   The template uses a mustache conditional:

   ```
   {{#components.length}}
     ...
   {{/components.length}}
   ```

   This will skip the nested section when length is `0`; i.e. it depends on the fact that one of the bad parts of JavaScript is that it interprets `0` as **false** in a boolean context. That is not the case for other languages like Java or Ruby. (Note that `0` being falsy probably came into JavaScript from Perl; that may also be the rational for it being falsy in Python, though Python appears to have really gone to town on falsy values).

   Rather than depending on the `length` property of `components`, it may be a better idea to add some kind of property like `hasKids` to each component. Note that compound components should have `hasKids` set to **true**, while leaf components should have `hasKids` set to **false**. Note that the latter is explicitly necessary for the same reason the original solution had leaf nodes with `components` set to `[]`; if not provided, then mustache's contextual lookup will find `hasKids` **true** in the parent leading to an infinite rendering loop.

   Though not required for this answer, a fixed version of the code is available.

2. When search terms include noise words in *Project 4*, those noise words are also highlighted in successful results.

   (a) Is this a bug or a feature?

   (b) What changes would you need to make in the overall setup for projects 3 and 4 to ensure that noise words are not highlighted in successful results? *15-points*

   The answers follow:

(a) When a user is conducting a search, the user is quite unaware of which words are noise words; in fact, the user may be totally ignorant of even the concept of noise words. Hence it would seem natural to the user that all the search-term words be highlighted in the results. Hence this behavior appears to be more a feature than a bug.

A technical case could be made for the fact that this behavior is a bug: highlighting the noise words is misleading since we never did actually search for them.

(b) In order to not highlight noise words, project 4 would need to know which words are noise words. We could hardcode the noise words into Project 4, but that would be a major DRY violation. A better alternative would be to have a web service return the list of noise words, or one to filter the search terms to remove noise words.

[Note that *Project 2* did provide a method **words()** which could be used to filter the search terms to remove noise words. This was not exposed as a web service in *Project 3* for simplicity and also as it was felt that the resulting behavior would be more a feature than a bug.]

3. The last route used for the *users application* has a comment *//must be last*. Discuss why this is so. What changes would you make to avoid such ordering requirements? *10-points*

   The reason that the specified pattern route `:id.html` must be last is that it matches earlier routes. So if it occurs earlier, those earlier routes would never be selected.

   One fix would be to ensure that the pattern does not match other routes. For example, changing the route from `${base}/:id.html` to something like `${base}/user/:id.html` would achieve this.

4. Discuss how you would design RESTful web services for recording and retrieving student grades for a typical university. Specifically, your discussion should include the following:

   - Details of the data model including support for different grading policies for different courses.
   - The web service API including choice of HTTP methods, URLs and parameters.

   You may regard any authorization requirements for the web services as orthogonal issues outside the scope of your design. *15-points*

   This question requires analyzing the requirements for a grading system and coming up with a web service API based on that analysis.

   The first concept which needs to be nailed down is the idea of a grade:

   (a) A grade can be a course grade which is understood by the university.

(b) A grade can be internal to a course. These may be grades assigned to individual assignments.

(c) The values for a grade can include the common letter grades, `A+`, `A`, `A-`, etc; a numeric value; options like `PASS` or `FAIL`.

So in general we will have a `Gradable` entity like a course, homework, paper or project. Associated with each `Gradable` will be will be a `GradingOption` which may be `Letter('A+', 'A', 'A-', ...)`, `Numeric(0, 100)`, `Numeric(0, 3)`, `Options('Pass', 'Fail')`, etc.

We will need some way of grouping `Gradable`'s together. This can be done by associating one or more categories with each gradable. Example categories used at the University level for categorizing course `Gradable`'s would be `Major`, `Minor` for categorizing courses which meet a major/minor requirement. Example categories used within a course would include `Homework`, `Project`, `Paper`.

We will need some kind of `GradeFormula` abstraction for combining grades together. One formula could take `GradingOption Letter` grades and compute a GPA. Another formula could take a set of `GradingOption Numeric` grades and compute the average after dropping the lowest grade.

We will also need the entities for a `CourseOffering` and a `Student`. These entities would be identified by some opaque id which could be derived from a database key. It could also be built based on the entity; for example, a `CourseOffering` id could be formed from the course number/section like `cs580w-01` combined with a semester to get an id like `cs580w-01_f18`.

There would be a many-to-many correspondence between `CourseOffering`'s and `Students`. This would lead to a basic tension in our data model; should we group `CourseOffering`'s under a `Student` or group `Student`'s under a `CourseOffering`. We should try to accomodate both.

For each `CourseOffering-Student` pair we would have a list of `Gradables`.

Our web services would be set up under two basic URLs: `/course-offerings` and `/students`. We would provide HTTP methods for accessing or adding entities at these URLs.

`GET` A `GET` to one of these URLs would result in a summary list of all the corresponding entities. Since the number of entities will likely go into the 1000's, we should allow filter query parameters like `/course-¬ offerings?year=2018` or `/students?year=2018&department=cs`.

`POST` A `POST` request to one of these URLs would create a new entity. The body of the request would be JSON containing the details of the entity being created.

The returned response will contain a `Location` header giving the URL for the newly created course offering.

For each of the above two URLs we would have methods for subordinate URLs based on the id of the course offering or student. For example:

`GET /students/:id` This would return all univerity level information including university level grades for the student specified by `:id`. We could use query parameters to specify exactly what information should be returned. Similarly for `/course-offerings/:id`.

`PATCH /course-offerings/:id` Update information for course offering specified by `:id`. Similarly for `/students/:id`.

`DELETE /course-offerings/:id` Remove course offering specified by `:id`. Similarly for `/students/::id`.

Note that the above URLs can be used for updating / accessing what is normally thought of as data like grades as well as meta-data like assigning categories to gradables or adding formulas.

The system should come with built-in meta-information for common categories and formulas. However, it should be possible to define additional meta-information beyond what is standard; some kind of plugin system can be used to provide these extensions.

5. Discuss which HTML control you would use for each of the following form controls:

   (a) A control which allows the user to provide the percentage of components which are defective.

   (b) A control which allows the user to select one-or-more of the courses which are currently being offered by the CS department.

   (c) A control which allows the user to enter a customer-service complaint.

   (d) A control which allows the user to enter in a BU B-Number.

   (e) A control which allows the user to select their favorite dessert from a predefined list of desserts.

   (f) A control which allows a US user to enter in a telephone number (which may be an international phone number).

   Your answers should be set up to maximally constrain the user's input to legal values and minimize the opportunity for error. You should use at most a single HTML control for each of the above and should provide the actual HTML code for each control. *15-points*

   (a) The percentage of components which are defective would be between 0 and 100. So assuming that an integer is acceptable for the percentage, something like:

```
<input name="percentDefective" type="number"
       min="0" max="100">
```

would be acceptable.

If the percentage does not need to be entered too precisely, then a range control may also be acceptable:

```
<input name="percentDefective" type="range"
       min="0" max="100">
```

The latter may be preferred if the rest of the form controls are all mousable.

(b) Two possibilities come to mind:

- Using checkboxes.
- Using a select box with the `multiple` attribute.

Given the relatively large number of courses, neither choice is ideal.

If checkboxes are used, then the courses should be formatted compactly. Something like:

```
<table>
  <tr>
    <td>
      <input type="checkbox" name="courses"
             value="110>
      CS 110
    </td>
    <td>
      <input type="checkbox" name="courses"
            value="140>
      CS 140
    </td>
    <td>
      <input type="checkbox" name="courses"
            value="210>
      CS 210
    </td>
   </tr<
   ...
</table>
```

A select box could be specified as follows:

```
<select name="courses" multiple>
  <option value="110">CS 110</option>
  <option value="140">CS 140</option>
  <option value="210">CS 210</option>
  ...
</select>
```

An argument against a select box is that many users are unfamiliar with the concept of using a select box to make multiple selections. If that is true for the target population, then a checkbox would definitely be the best choice.

(c) The only possible choice would be a `textarea`.

```
<textarea name="complaint" rows="10"></textarea>
```

(d) An input tag with a regex to constrain the input:

```
<input name="bNumber" pattern="B\d{8}"
       placeholder="B12345678">
```

(e) A select box is probably appropriate:

```
<select name="favoriteDessert">
<option value="bananaSplit">Banana Split</option>
<option value="chocolateCake">Chocolate Cake</option>
  ...
</select>
```

(f) A telephone number is always problematic. Complexities include:

- International telephone numbers and the unfamiliarity of most US users with the standard + country-code prefix.
- The possibility of needing to enter an extension, usually after the letter `x`.
- Different delimiters used for separating different parts of the number. For example, is a US phone number to be entered as (607) 777-2000 or as `607-777-2000`, or as `607 777 2000`?

The best which could be done would be to use an `input` control with `type=tel` and use a reasonably restrictive pattern. Given the variety of possible inputs, a placeholder may not be possible but the text around the control should describe expected telephone number formats.

```
<input name="phone" type="tel"
       pattern="\+?[\d\-\(\)\s]+(x\s?\d+)?">
```

After the user has provided the input, it will be necessary to run fairly smart code to parse the entered number heuristically and validate it.

If there is context elsewhere to figure out what country phone number the user is about to enter, then it may be possible to provide a more constrained pattern as well as an appropriate placeholder. For example, if we know definitely that the user is about to enter a US phone number without an extension:

```
<input name="phone" type="tel"
       pattern="\(\d{3}\)\s\d{3}\-\d{4}"
       placeholder="(123) 456-7890">
```

6. Discuss how you would serialize an abitrary JavaScript object **hierarchy** to HTML using mustache. You may assume that the values of object properties are either primitives or objects or arrays. *15-points*

   There are two parts to this question:

   (a) The exact HTML constructs used for the serialization:

   - Objects are serialized as `<dl>...</dl>` definition lists with property names embedded within `<dt>...</dt>`, and the recursive serialization of property values embedded withing `<dd>...</dd>`.
   - Arrays could be serialized as ordered lists using `<ol>...</ol>`. If the order of the array elements is not regarded as significant, then they could be serialized using `<ul>...</ul>`. The recursive serialization of the array elements would go into `<li>...</li>`.
   - Primitives would be serialized using their string representations.

   (b) How to achieve the serialization using mustache:

   This would be very similar to the `recursive.js` example from the first question. It would be critical to create a view model to get around the limitations of mustache.

   Since mustache can only loop through arrays and not objects, it is necessary to convert all objects to arrays.

   All JavaScript values would be converted to uniform objects with the properties: `isArray`, `isObject`, `isKeyValue` and `isPrimitive`

(only one of these four properties would be true), as well as a `value` property. The latter would be set as follows:

- For a primitive it would be the string value of the primitive.
- For an array it would be an array of the view models of the element values.
- For an object it would be an array of objects with three keys: `isKeyValue` with the value **true**, `key` and `value` with the latter giving the view model of the property with name `key`.

Using this view model it should be possible to set up a mustache template with four mutually exclusive sections (chosen based on which of `isArray`, `isObject` `isKeyValue` or `isPrimitive` is true). The template would use a recursive partial to render arbitrarily nested objects.

7. Discuss the validity of the following statements. What is more important than whether you ultimately classify the statement as **true** or **false** is your justification for arriving at your conclusion. *15-points*

   (a) It is more efficient to send JSON rather than XML over a network since JSON is a more compact representation than XML.

   (b) If an attempt to create a resource using `PUT` hangs, then it is ok to make a second attempt.

   (c) If an attempt to create a resource using `POST` hangs, then it is ok to make a second attempt.

   (d) If a HTML document has a syntax error, then an attempt to render it in a browser will result in an empty page.

   (e) REST is a framework for building web services.

   The answers follow:

   (a) JSON is definitely more compact than XML. Consider the equivalent fragments:

```
<person>
  <name>
    <first>Bart</first>
    <last>Simpson</last>
  </name>
</person>
```

   versus:

```
{ "person": {
    "name": {
```

```
            "first": "Bart",
            "last": "Simpson"
        }
    }
}
```

If we count the number of characters (ignoring whitespace), the above XML clocks in at 69 characters while the JSON clocks in at 53 characters. The difference is likely to be more substantial for more highly nested documents where the end tags in the XML would add up.

So if our analysis was complete, the answer definitely be true. But our analysis is not complete since it is often the case that network transmissions are compressed and the cost of XML end tags is likely to disappear. For example, running gzip on the above fragments without whitespace resulted in 77 and 73 characters respectively; so the XML compressed to almost the same size as the JSON. (Note that compressed size is larger than uncompressed size because of the compression overhead which will be insignificant for larger files).

So the statement is **true** if compression is not being used; and tends to be **false** if compression is used.

(b) Since `PUT` is defined to be idempotent, then it is ok to make an additional attempt. So the statement is **true**.

(c) Since `POST` is not defined to be idempotent, it is not ok to make an additional attempt. So the statement is **false**.

(d) Browsers tend to be very forgiving of syntax errors; so attempting to render a syntactically incorrect page is uniikely to result in an empty page.

Note that this was not true for early attempts at implementing XHTML (which was one reason why XHTML failed).

If we regard the term HTML to refer to non-XHTML HTML, then the statement is definitely **false**.

(e) The answer is it depends on what is meant by the term *framework*.

- If *framework* is used in a technical sense as a program in which code can be embedded then the statement is definitely **false**.
- If *framework* is used in a more general sense to refer to *a basic conceptional structure*, then the statement is definitely **true**.