1 CS 480W/580W Final Solution

Date: Dec 14, 2018 **Max Points**: 100
Time: 120 minutes.
Open-book, open notes; **no electronic devices**

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.

Background:

1. A student needed to extract the body of a HTTP request. The student observed that the HTTP headers in a sample request contained 271 characters and proceeded to use the following code to extract the body of a `request`:

   ```
   const requestBody = request.slice(271);
   ```

   What is wrong with this approach? How would you fix the problems? (Actual code is not necessary, it is sufficient to clearly describe the problem and the suggested fix). *10-points*

   The problems are the following:

   (a) Strange magic numbers like 271 within code are **always** a bad idea as a reader looking at the above code segment would have absolutely no idea of what is going on. If the code was changed to something like:

   ```
   const HEADERS_LENGTH = 271;
   const requestBody = request.slice(HEADERS_LENGTH);
   ```

   the name of the constant at least provides some idea of the programmer's intentions.

   (b) The major problem with this approach is that it is completely broken. The headers may happen to be 271 characters for one particular request from a particular version of a browser running on a specific OS; it will likely be a completely different number for a different browser.

   An approach which would work is to follow the HTTP standards which document that an empty line separates the HTTP headers from the body. So the code should look for that empty line in order to determine the start of the body.

   [Some form of the above statements are sufficient to get full credit for this question.]

The standard documents that lines be terminated by CRLF \r\n. However, applying Postel's principle, we would look for the empty line using a regex like /\n\r?\n/ and obtain its offset. So something like:

```
const match = request.match(/\n\r?\n/);
if (!match) {
  //error handling
}
else {
  const bodyIndex = match.index + match[0].length;
  const requestBody = request.slice(bodyIndex);
  ...
}
```

2. A web service is called using **doService**(url, succ, err) where:

   url Specifies the URL for the web service.

   succ This is a callback function of one argument which is called with the result of the web service when the service succeeds.

   err This is a callback function of one argument which is called with an error object when the service fails.

   Write a wrapper function **wrappedService**(url) which wraps the above **doService**() so that the wrapper can be called as follows:

```
  try {
    const result = await wrappedService(url);
    //code to handle result
  }
  catch (err) {
    //code to handle web service error
  }
```

   Provide the code for **wrappedService**(). *15-points*

   Have the wrapper return a promise with the web service callbacks delegating to the arguments to the executor function:

```
async function wrappedService(url) {
  return new Promise((resolve, reject) => {
    doService(url,
              (result) => resolve(result),
              (err) => reject(err));
  });
```

```
    }
```

3. Given the following attempt at a `react.js` component for a Binghamton University **B-Number**:

```
01    class BNumber {
02      constructor(props) {
03        this.state = {
04          input: '',
05          error: '',
06        };
07      }
08
09      onBlurHandler(event) {
10        const input = this.state.input;
11        if (!input.match(/^B\d+/)) { //validate
12          this.state.error = 'invalid B-number'
13        }
14      }
15
16      render() {
17        return (
18          B-Number:
19          <input onBlur={this.onBlurHandler}>
20          <br>
21         <span class="error">{this.state.error}</span>
22        );
23      }
24
25    }
```

You may assume that all necessary libraries have been included.

Identify bugs and inadequacies in the above implementation of `BNumber`. *15-points*

Outright bugs:

- Line `01`: When an ES6 class is used to implement a `react.js` component, the class should extend `React.Component`.
- Line `03`: The constructor should call `super(props)`.
- Line `10`: The blur handler refers to `this`, but when called by the DOM, `this` will be set to the HTML widget on which the handler was registered. However, we need to have `this` set to point to the react component.

  This can be achieved by adding the line

```
this.onBlurHandler = this.onBlurHandler.bind(this);
```

to the constructor.

- Line `10`: `state.input` is never updated, so it will always be empty.

  One fix would be to remove it from the `state` and have its value accessed directly using `event.target.value` (the `event` will need to be added as an argument to the handler). However, this would violate the *single source of truth* react principle: some portions of the state would be maintained within the component and other portions maintained in the DOM.

  A better fix would be to add a handler for `onChange` and have that handler update `state.input`, maintaining all state within the component.

- Line 12': The component state should never be set directly; it should only be set using `setState()`.

- Lines `19` and `20`: JSX expressions consist of well-formed XML. Hence the `<input...>` element should be `<input.../>` and the empty `<br>` element should be written as `<br/>`.

- Line `18`: React only allows a single JSX element embedded within JavaScript, not a sequence of adjacent elements. Hence those expressions should be wrapped within a top-level element like a `<div>`.

Inadequacies include the following:

- For better accessibility, the `B-Number:` text should be associated with the `<input>` widget. This can be done by wrapping both within a `<label>` element.

- The validation regex checks for start of the string using `^` but does not check for the end of the string allowing garbage after a correct B-number; this can be fixed by adding a `$` at the end of the regex.

- The validation would fail if there were leading whitespace characters or (with the above fix) trailing whitespace characters. This can be avoided by using `trim()` on the value before matching it with the validation regex.

- The error message could be more specific by specifying the entered value for which the validation failed.

Identifying around 5 of these problems should be sufficient to get full credit for this question.

4. During a code review, a colleague flags some lines from your code for a react.js component as constituting a major security flaw:

```
...
validate() {
  ...
  if (!input.match(...)) { //validate user input
    this.setState({error: `bad input ${input}`});
  }
  ...
}

render() {
  return (
    ...
    {this.state.error}
    ...
  );
}
```

Your colleague claims that because you are permitting unvalidated, unescaped user input to be added via the error message to the HTML page, a cracker can craft an input which allows execution of arbitrary JavaScript on your web page. Explain why your colleague is wrong. *10-points*

Your colleage is correct in having her security antennae twitching when she thinks she sees user input being rendered unchanged on the HTML page. However, where she is wrong is in claiming that the user input is being rendered unescaped: all HTML metacharacters within JavaScript strings in JSX are always escaped. Hence it is safe for you to simply render the unvalidated user input in the error message without explicit escaping.

[The default escaping behavior of react and other frameworks helps prevent XSS attacks. The flip side is that it makes it harder to produce dynamic HTML, as when highlighting search terms within search results in Project 5.]

5. You are given a JavaScript list `gameScores` of combined scores of NBA games where each element of the list has keys `teams`, `date` and `score` and is of the form:

```
{
  teams: Teams, //string of form WinningTeam-LoosingTeam
  date: Date,   //string of form YYYY-MM-DD
  score: Score, //integer giving sum of both team scores
}
```

Example data:

```
const GAME_SCORES = [
  { teams: 'Pistons-Nuggets',
    date: '1983-12-13',
    score: 370
  },
  { teams: 'Pistons-Lakers',
    date: '1950-11-22',
    score: 37
  },
  { teams: 'Mavericks-Blazers',
    date: '2018-12-04',
    score: 213
  }
];
```

(a) Critique the above data representation.

(b) Write a function `renderScores(gameScores)` which uses `mustache¬ .js` to render `gameScores` into a HTML table, such that:

- The return value of the function should be a string containing the rendered HTML with top-level element `<table>`.
- The table should have a 3-column heading row with the columns labelled **Teams**, **Date** and **Score** respectively.
- The heading row must be followed by data rows, one for each item in `gameScores` with its `Teams`, `Date` and `Score` in the appropriate column.
- The table must have a CSS class of `gameScores`.
- if `Score < 150`, then the data row should have its CSS class set to `low`.
- if `150 <= Score < 250`, then the data row should have its CSS class set to `mid`.
- if `Score >= 250`, then the data row should have its CSS class set to `high`.

For example, given the above example data, the call **renderGameScores**`(GAME_SCORES)` should return:

```
<table class="gameScores">
<tr><th>Teams</th><th>Date</th><th>Score</th></tr>
  <tr class="high">
    <td>Pistons-Nuggets</td>
    <td>1983-12-13</td>
    <td>370</td>
  </tr>
```

```
    <tr class="low">
      <td>Pistons-Lakers</td>
      <td>1950-11-22</td>
      <td>37</td>
    </tr>
    <tr class="mid">
      <td>Mavericks-Blazers</td>
      <td>2018-12-04</td>
      <td>213</td>
    </tr>
  </table>
```

modulo whitespace.

You may assume that the `mustache` module has been `require`'d and is available using identifier `mustache`. *20-points*

During the exam it was announced that `renderScores` ≡ `renderGameScores`.

(a) The problem with the representation is that valuable information is encoded into strings and it will be necessary to parse strings in order to extract the information.

- To extract the name of the winning and loosing teams it will be necessary to do something like `teams.split('-')`. This would be problematic if a team name contains a hyphen. It would be better to have a separate field for each team.
- The year, month and day of the game-date are buried within a string. It would be better to use a JavaScript `Date` object.

(b) Since mustache is logic-less, it is necessary to perform the `low-mid-high` computations outside the template. So the function could look like:

```
const LO = 150, MID = 250;

const TEMPLATE = `
  <table class="gameScores">
 <tr><th>Teams</th><th>Date</th><th>Score</th></tr>
    {{#gameScores}}
      <tr class="{{klass}}">
     <td>{{teams}}</td><td>{{date}}</td><td>{{score}}</td>
      </tr>
    {{/gameScores}}
  </table>
`;

function renderGameScores(gameScores) {
  const augmentedGameScores =
        gameScores.map(function(gameScore) {
          const {score} = gameScore;
          const klass =
            score < LO ? 'low'
            : score < MID ? 'mid' : 'high';
          return { ...gameScore, klass: klass };
        });
      const view = {
        gameScores: augmentedItemScores
      };
      return mustache.render(TEMPLATE, view);
    }
```

Runnable code; requires mustache (obviously not required for submitted exams).

6. Assuming that a HTML page contains zero-or-more tables formatted as per the previous question:

   (a) Write a jQuery selector to return a jQuery collection which contains all the `<td>` elements containing `high`-level scores from all such tables on the page.

   (b) Write a JavaScript function `highScoreAverage()` whic uses jQuery to average all the `high`-level scores on the page. You may assume that `jQuery` is accessible as usual in the shortcut `$` variable.

   Hint: A jQuery collection `$list` of jQuery objects can be converted to a JavaScript array of DOM objects using `$list.toArray()` and the HTML content of a DOM object `dom` can be extracted using the property `dom.innerHTML`. *15-points*

   The answers follow:

   (a) We would first select the table using its class `gameScores`, then select all descendent rows having class `high` and then select the 3rd `td` child of all matching rows. The selector would be:

   `"table.gameScores tr.high :nth-child(3)"`

   (b) Using the above selector we can write `highScoreAverage()`:

```
    function highScoreAverage() {
      const scores =
        $("table.gameScores tr.high :nth-child(3)").
        toArray();
      const sum =
        scores.
        reduce((acc, v) =>
                   acc + Number(v.innerHTML), 0);
    return (scores.length === 0) ? 0 : sum/scores.length;
}
```

Runnable code; (obviously not required for submitted exams).

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) Mustache will always escape any HTML characters in rendered strings.

   (b) The cache time specified for images should be the same as that for their containing HTML page.

   (c) It is ok for a client to attempt to `DELETE` the same resource multiple times.

   (d) If the propagation of a DOM event is stopped in its bubble phase, the event will never enter its capture phase.

   (e) If a promise is rejected before a `catch` handler is attached to the promise, then the `catch` handler will not be run.

   The answers follow:

   (a) Mustache will escape HTML characters in rendered strings if those strings are rendered within double-braces but not within triple-braces. Hence the statement is **false** in general.

   (b) In general, an image may be referenced by multiple HTML pages with different cache times. So this would not be possible in general.

   Usually, the following pattern is used to maximize the caching of static assets like images: each version of an image is specified using a distinct URL having an "infinite" cache time. When an image is changed, the containing HTML pages are changed to refer to the URL corresponding to the new version.

   Hence the statement is **false**.

   (c) Since `DELETE` is idempotent, the statement is **true**.

   (d) Since the capture phase precedes the bubble phase, the capture phase will have run before the event propagation is stopped in its bubble phase. Hence the statement is **false**.

   (e) The promise captures the settlement of the promise; specifically, when the promise is rejected, it captures the corresponding error object. Hence `catch()` handlers for a rejected promise will always be run, irrespective of whether those handlers are attached before or after the rejection. Hence the statement is **false**.