causata

Front-end/JavaScript Developer

There are two sections to this test:

- an HTML/CSS section, covering standard markup problems
- a JavaScript section, covering more programmatic questions

There are no off-the-shelf solutions; there is also no single solution to any given problem. Be creative. We don't expect you to spend hours of your time coming up with answers, and partial responses may be accepted for discussion.

causata.com/about/careers

Part 1:

Create an HTML document containing a single element, that resizes with the browser window (the *white* element in *Figure 1*, below).

This element should contain three side-by-side child elements, all of which will contain complex markup (the *red*, *green* and *blue* elements in *Figure 1*, below). These elements are vertically aligned in the middle of the variable-height parent element.

- The *red* element on the left measures 100px x 200px, and is left-aligned within the parent.
- The *blue* element on the right measures 100px x 300px, and is right-aligned.
- The *green* element in the middle is 400px high, and of variable width. When the parent is resized, the *green* element adapts, always keeping a 10px distance away from its siblings. However, it must maintain a minimum 100px width, and 10px margins, as illustrated in *Figure 2* and *Figure 3*, below.

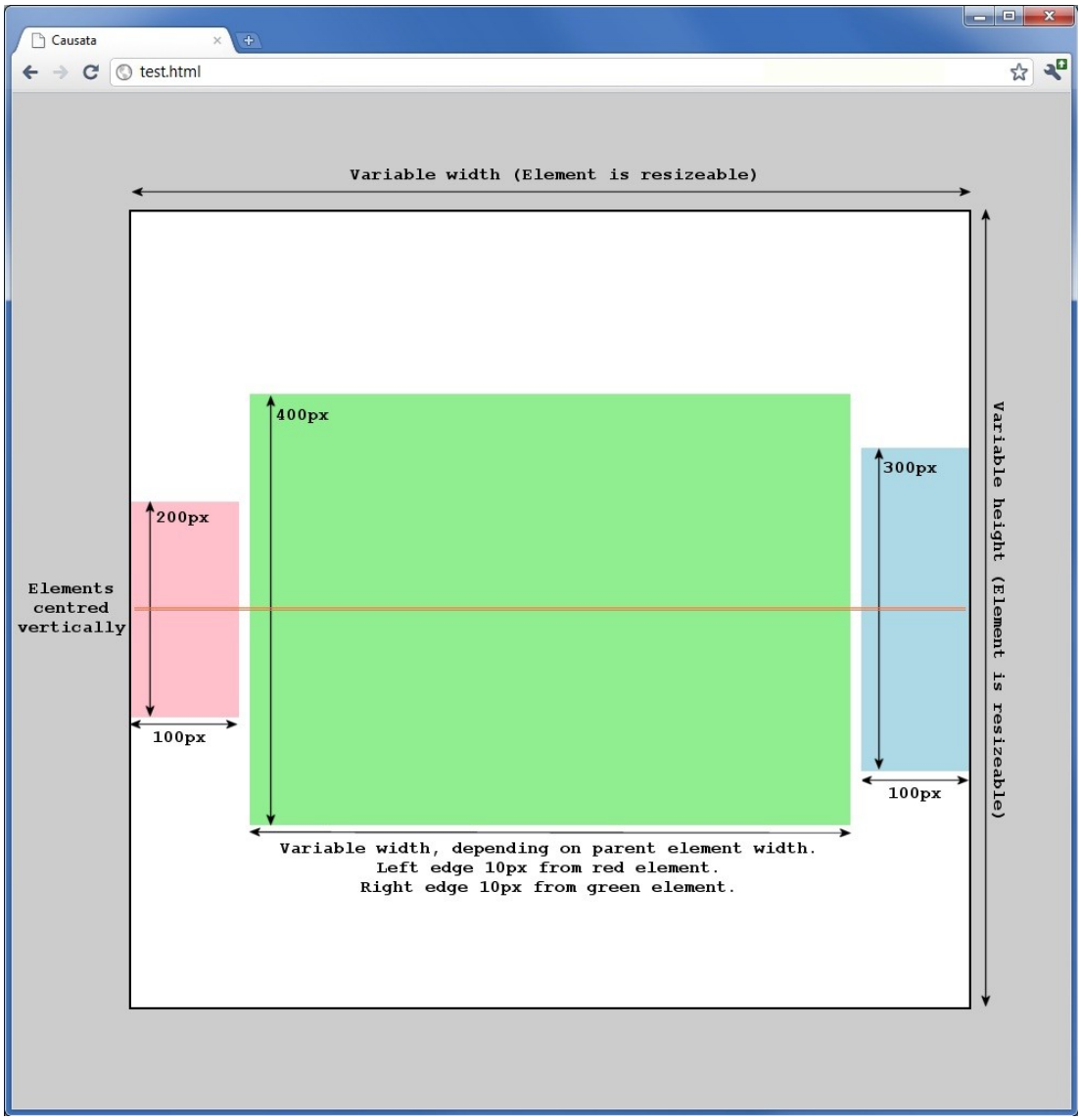

Figure 1: An example highlighting the specification for a simple three-element layout.

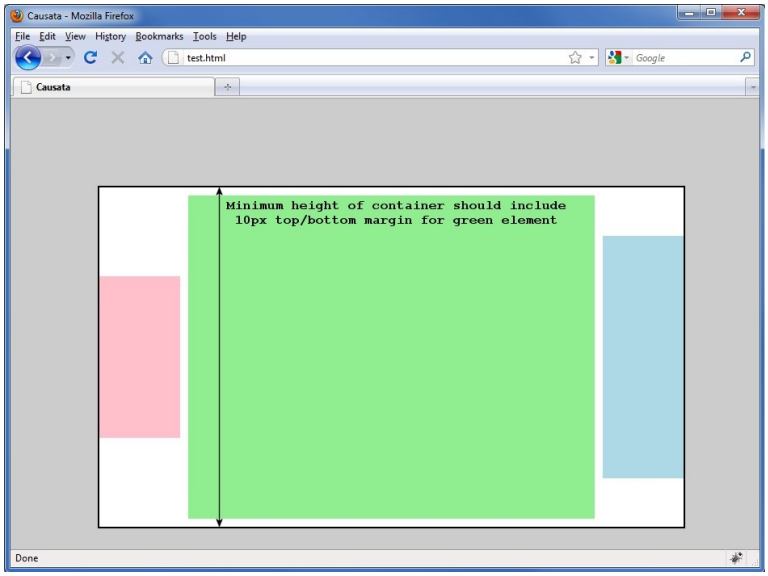

Figure 2: An example showing how the minimum height of the layout should behave.

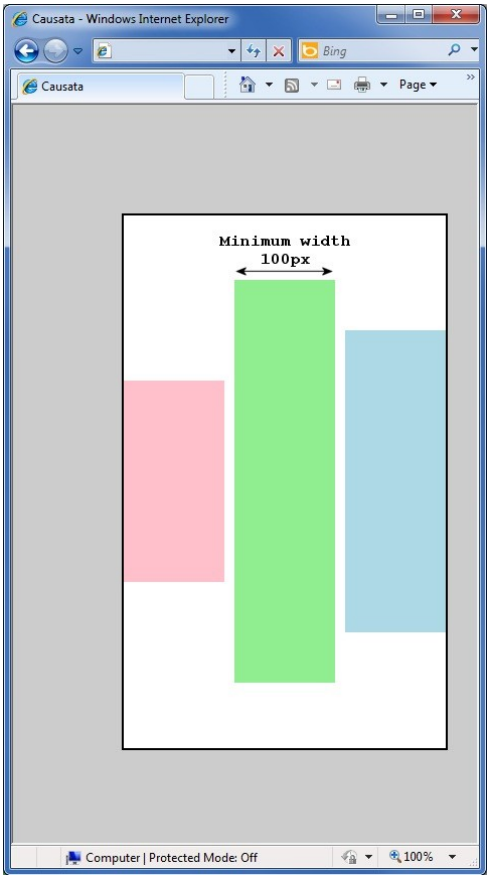

Figure 3: An example showing how the minimum width of the layout should behave.

Use only standard HTML and CSS – JavaScript and IE CSS expressions are *not* allowed. Similarly, HTML tables are *not* allowed. Your solution must work in IE8+, Firefox 3.6+, Chrome and Safari. The same markup must work in all supported browsers; conditional comments, CSS hacks, proprietary tags, attributes and properties are *not* allowed.

Part 2:

Using your layout from Part 1, implement—using HTML, CSS and/or JavaScript (as you see fit)—the design provided in *Figure 4*, below. Do not take this necessarily as a challenge; prioritize your time and focus on a suitable solution. We are not expecting pixel perfection, but instead a realistic answer to a real-world scenario.

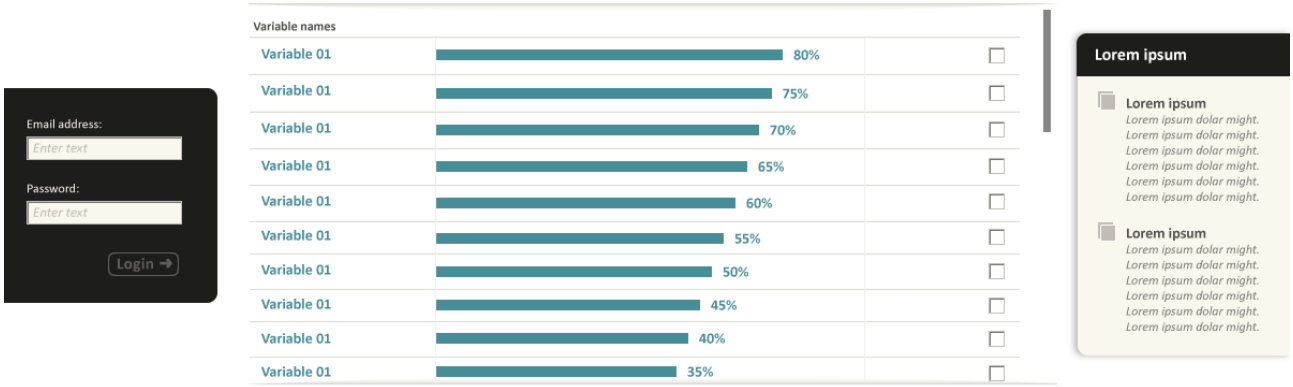

Figure 4: An overview of the final design requirements.

Discuss, briefly:

- Which design details are difficult to implement
- Which design details are impossible to implement
- As regards the designs themselves:
  - Do they provide you with enough information to facilitate implementation? How would you change them? How might you seek confirmation or feedback?
  - Are they good designs? That is, in your opinion, do they contain design errors? If they were to be implemented as is, would they enhance or hinder the user experience?
- What browser-specific issues arise from the designs?
- What compromises or changes would you make to facilitate cross-browser implementation?

The Log In Form:

Note that by default the 'Log In' button is in a disabled state, and the inputs have placeholder text.

Add HTML, CSS and/or JavaScript to display an "active" state for the 'Log In' button when the user enters text in the inputs (see *Figure 5*).

Also include a "hover" state for the 'Log In' button (*Figure 6*).

If the user input is invalid, the relevant input element should be styled to identify it as containing invalid input, and the 'Log In' button should remain disabled (*Figure 7*).

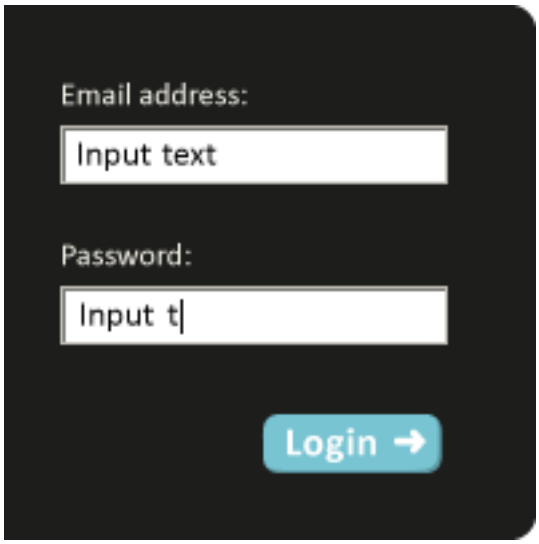

Figure 5:The 'Log In' button active state.

Figure 6:The 'Log In' button hover state.

Figure 7:The 'Log In' form in an error state.

The right side box:

This box should have space for a title, and generic content. There are no specific requirements as regards interaction.

The Data Table:

In the table, create a hover state for the rows (see *Figure 8*). Allow a custom checkbox to be 'clicked' without selecting the row (*Figure 9*), while clicking on the row anywhere other than on the checkbox should 'select' the row (*Figure 10*). Multiple rows may be selected, regardless of the checkbox state (*Figure 11*).

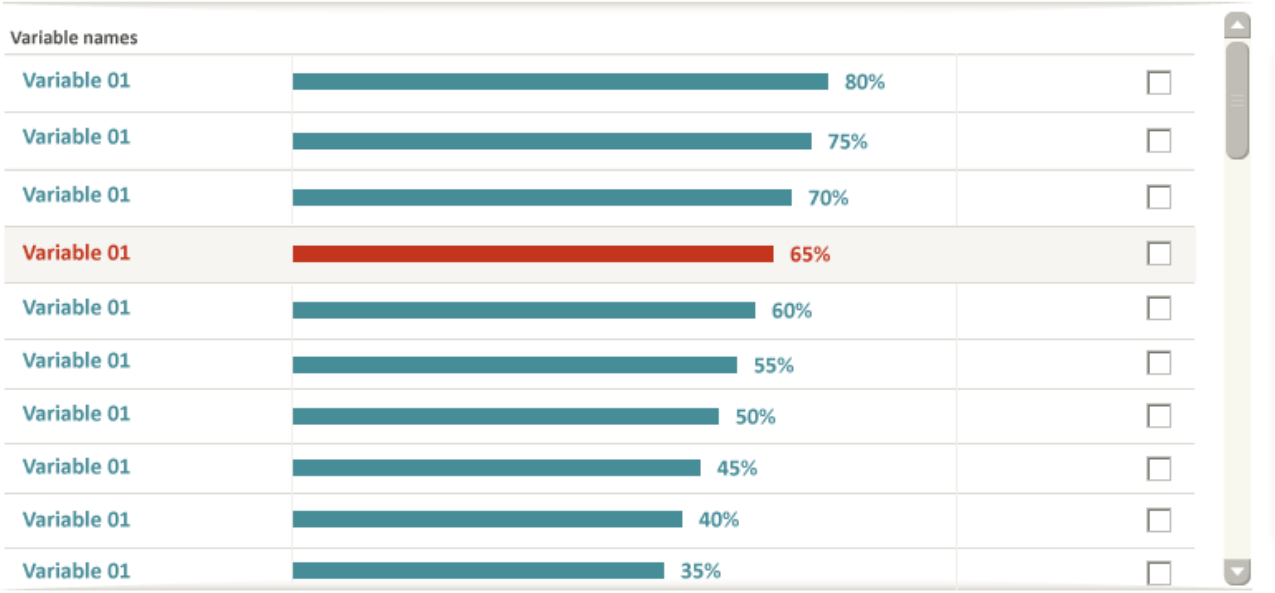

Figure 8: Rows should have a hover state.

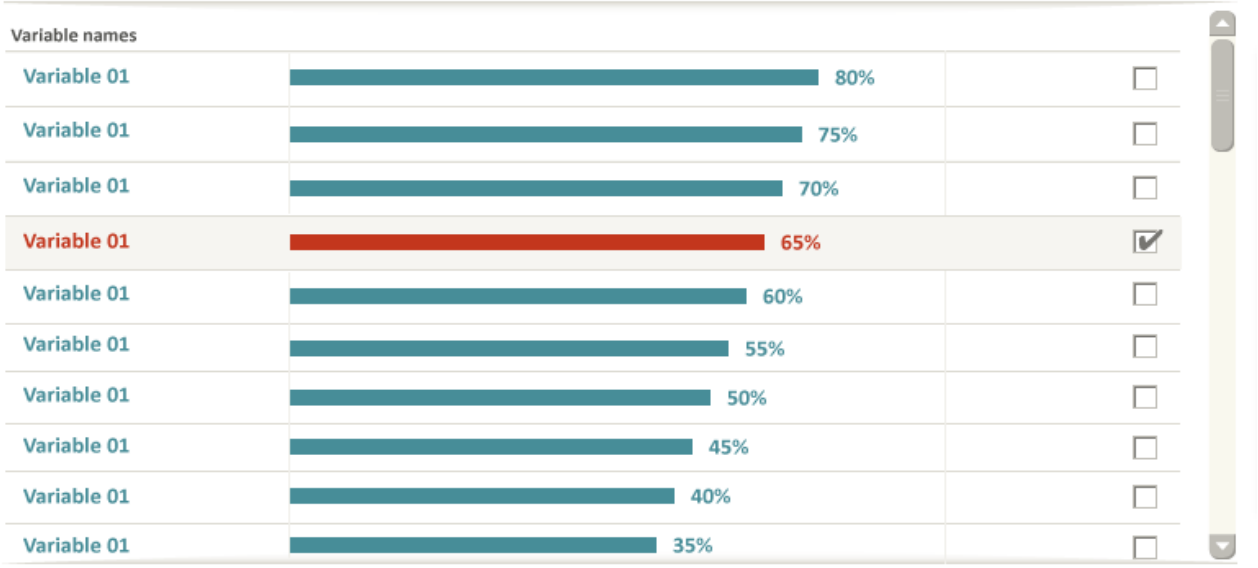

Figure 9: Custom checkboxes can be selected.

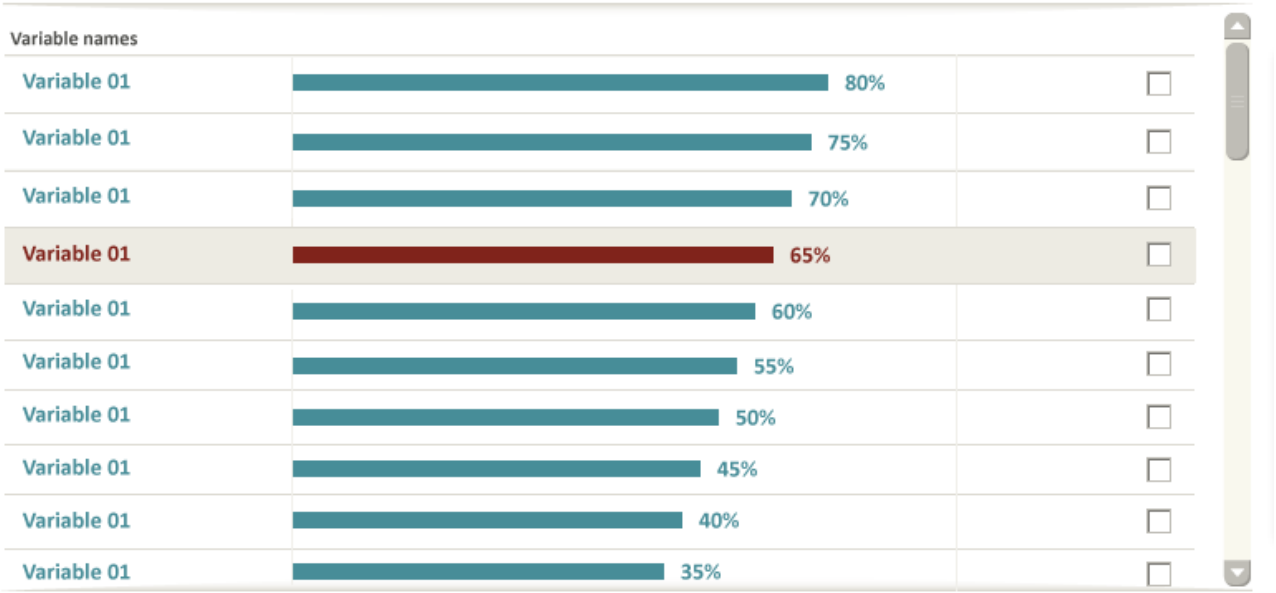

Figure 10: Clicking a row other than on the checkbox should 'select' the row.

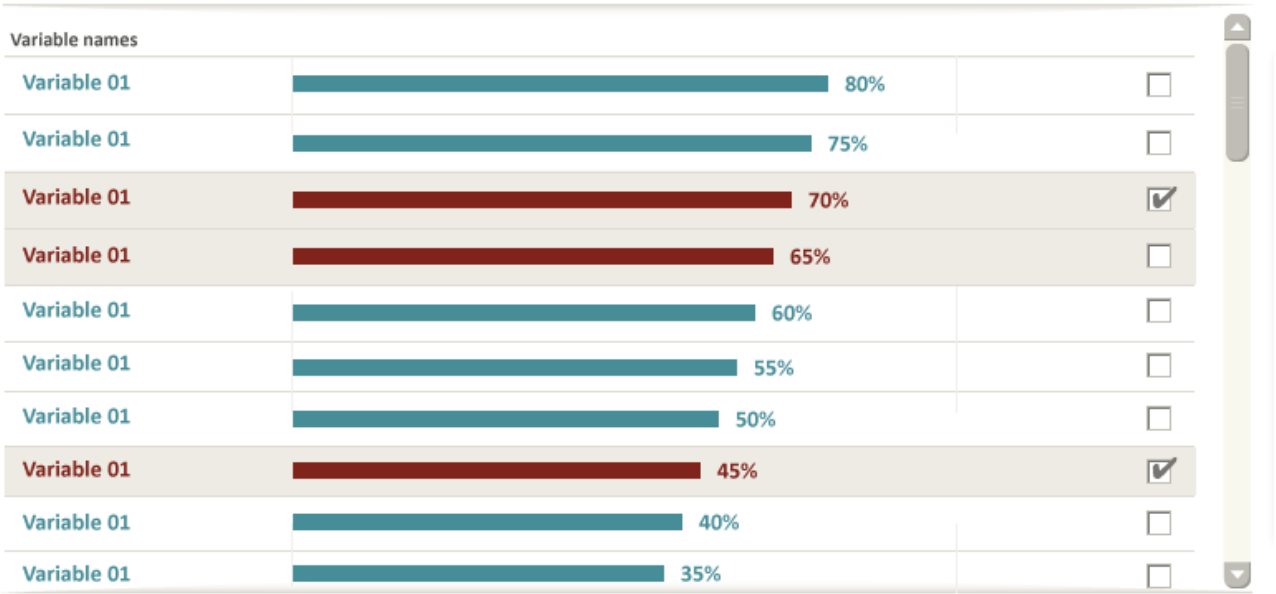

Figure 11: Mulitple rows may be selected, regardless of the checkbox state.

Part 1:

Sample 1 below contains a JavaScript implementation of a basic Component class, with an instantiation of it. Unfortunately, it doesn't work. Rewrite it so that the code will run properly.

```
var Component = function (config) {
    for (property in config) {
        this[propety] = config[property]
    }
}

var list = Array (
    "Item 1",
    "Item 2",
    "Item 3",
)

var instance = Component(id: "XF-254", list: list);
```

Sample 1: Code for a basic Component class, and an instantiation of it.

Part 2:

Change the code so that the `Component` class implements the Publish/Subscribe pattern. Clients should be able to register for notifications using simple keys, e.g. `propertyChanged`, `user.loggedOut`, or any other event that might make sense in an application.

Part 3:

Extending your answer to Part 2, suppose we now want to allow clients to receive notifications even if they registered themselves *after* such notification had been broadcast. Provide a simple implementation, noting any potential performance concerns.

Part 4:

Define your own `CustomComponent` that extends `Component`. It should have a `setValue` method which broadcasts an event when it is called, specifying the old and new values.

Bonus points:

How might your implementation change to allow for different base classes to implement the Publish/Subscribe pattern? How might your implementation change to support wildcard tokens in the message key, e.g. `app.*.log`? How might your implementation change to support a limited number of notifications for a given message?