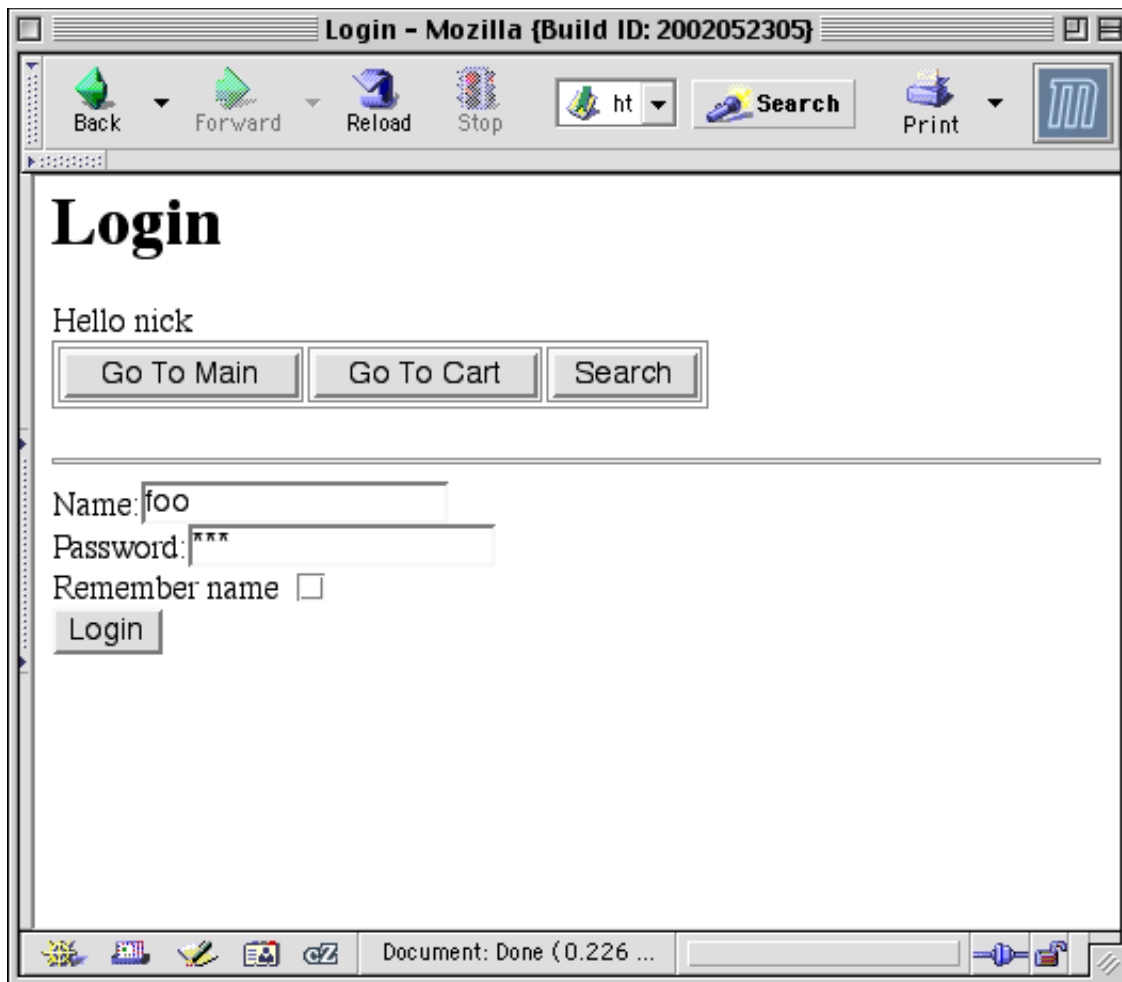


HW4b Amazon.edu

This handout describes parts B and C of the Amazon homework, which add cookie, session, and JavaScript features to the basic structure in part A. Taken all together, HW4 combines many of the technologies we have seen into a single program. The whole thing is due midnight ending Tue June 4th.

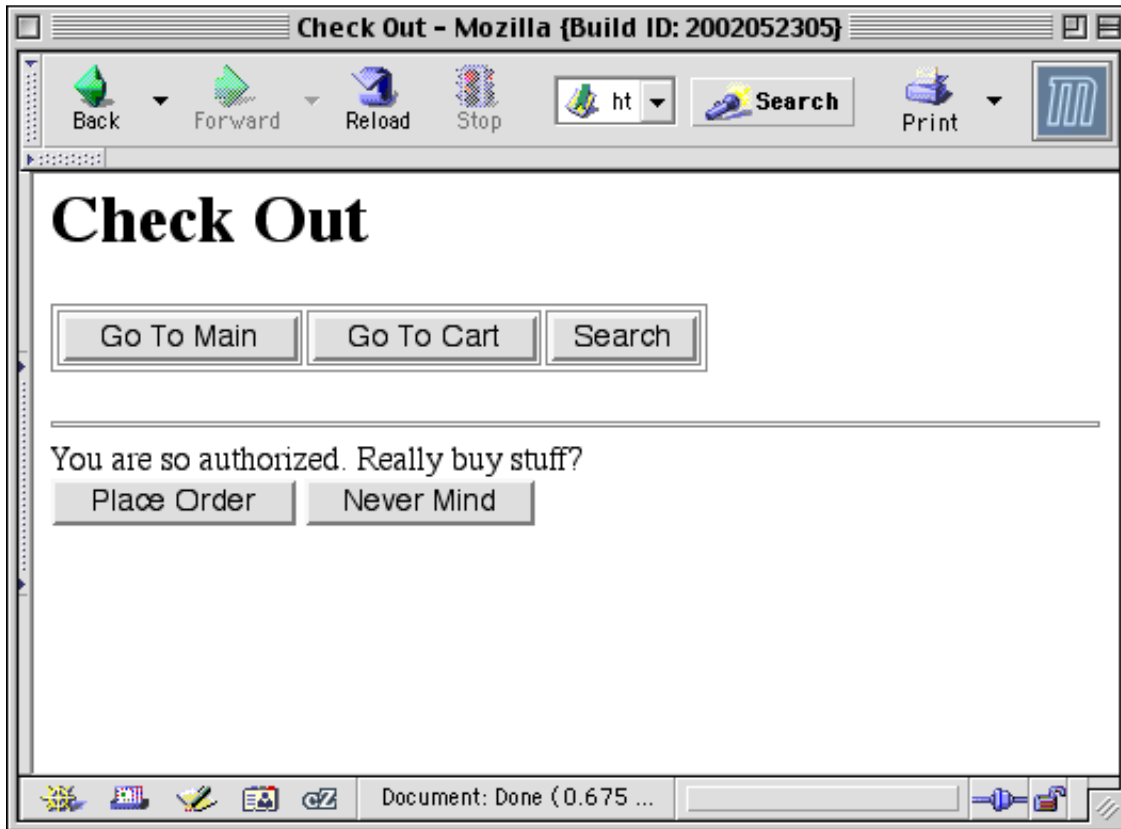
Part B -- Checkout

Part B will focus on the "checkout" feature using cookies. There are three pages in the checkout process, which we'll call c1, c2, and c3. In the simplest case, the user is lead through the 3 pages in order. The Check Out button from the cart page leads to c1 where the user can put in their username and password...

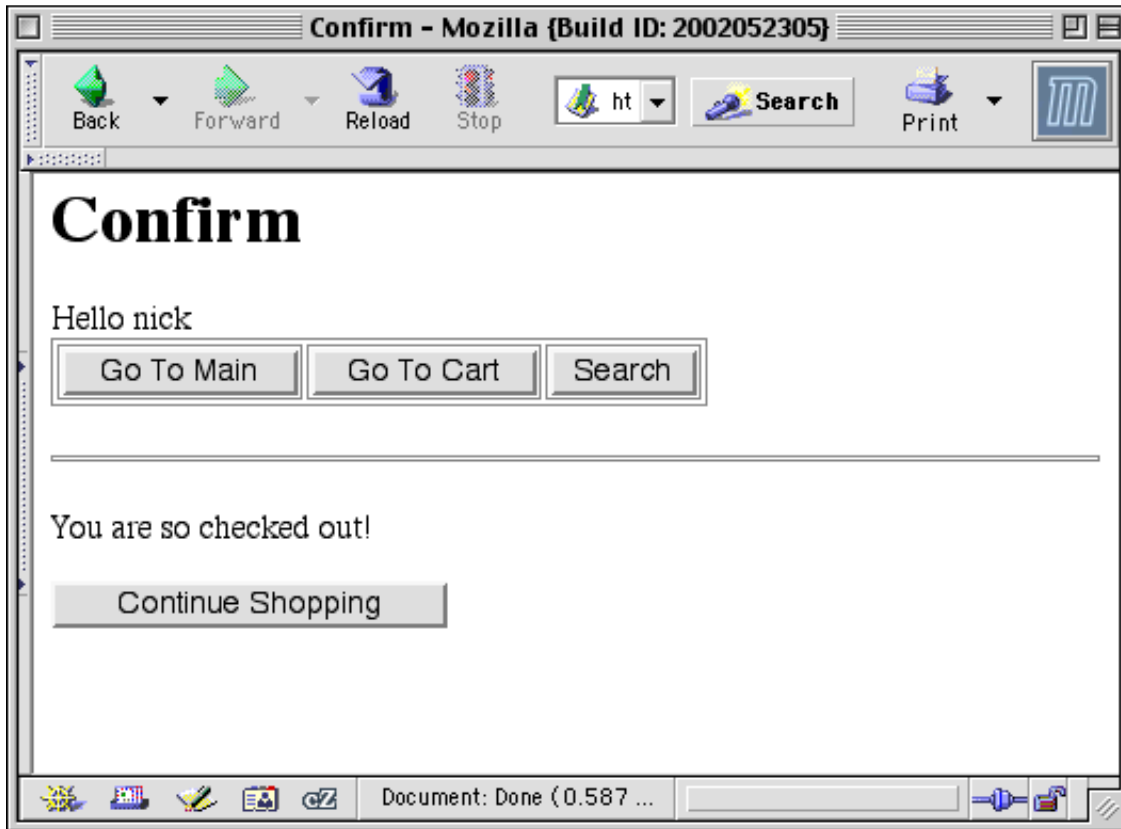


If the user tries to login, but the password is not correct, return page c1 again. At first, just let the password be the same as the username -- so if user "foo" logs in with password "foo", the password is good.

The c2 page confirms that the username and password are correct (the user is "authorized"), and the user can now choose to complete the purchase...



Placing the order leads to c3 which confirms that the order has happened and allows the user to return to the main page...



JSP

Implement the three pages, c1, c2, and c3, as JSPs. They are mostly just static HTML, so it's cleaner to code them as JSPs rather than clogging up the servlet code with lots of HTML.

Here are a few other cases for the check out pages...

Authorized

Suppose the user is seeing c2 -- they are authorized. Instead of checking out, the user returns to the main page and does some more shopping. When, later, they hit the Check Out button, the servlet should know that they are already authorized, and so skip c1 and go right to c2. Whether or not the user is authorized should be stored in the server side session, not in a hidden field or cookie.

Reset On Confirm

If the user confirms the order and so gets to c3...

- Their cart should be emptied out, so going to the cart page should show no books

- Their authorization state should be set back to not authorized, so going to c1 should ask for the password again. (This is for the "library" problem.)

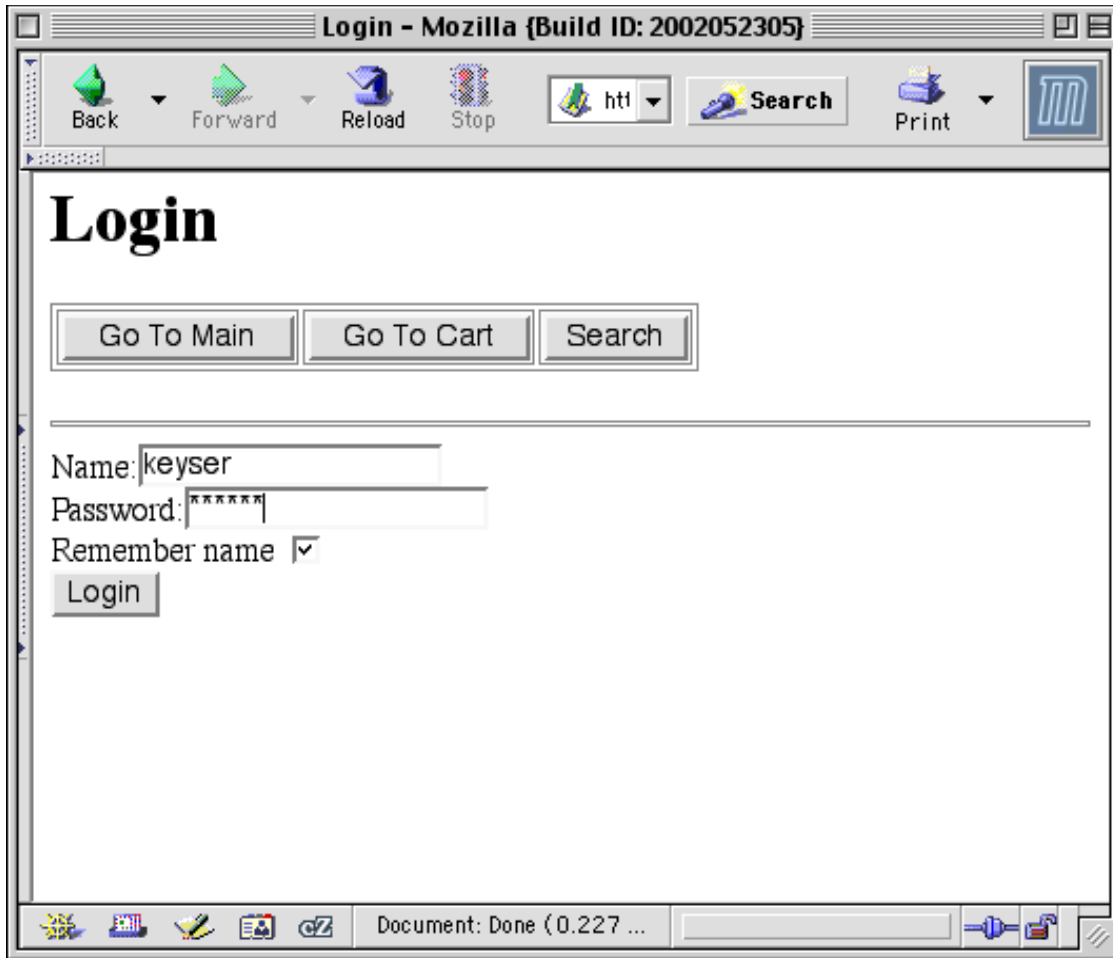
Remember Name

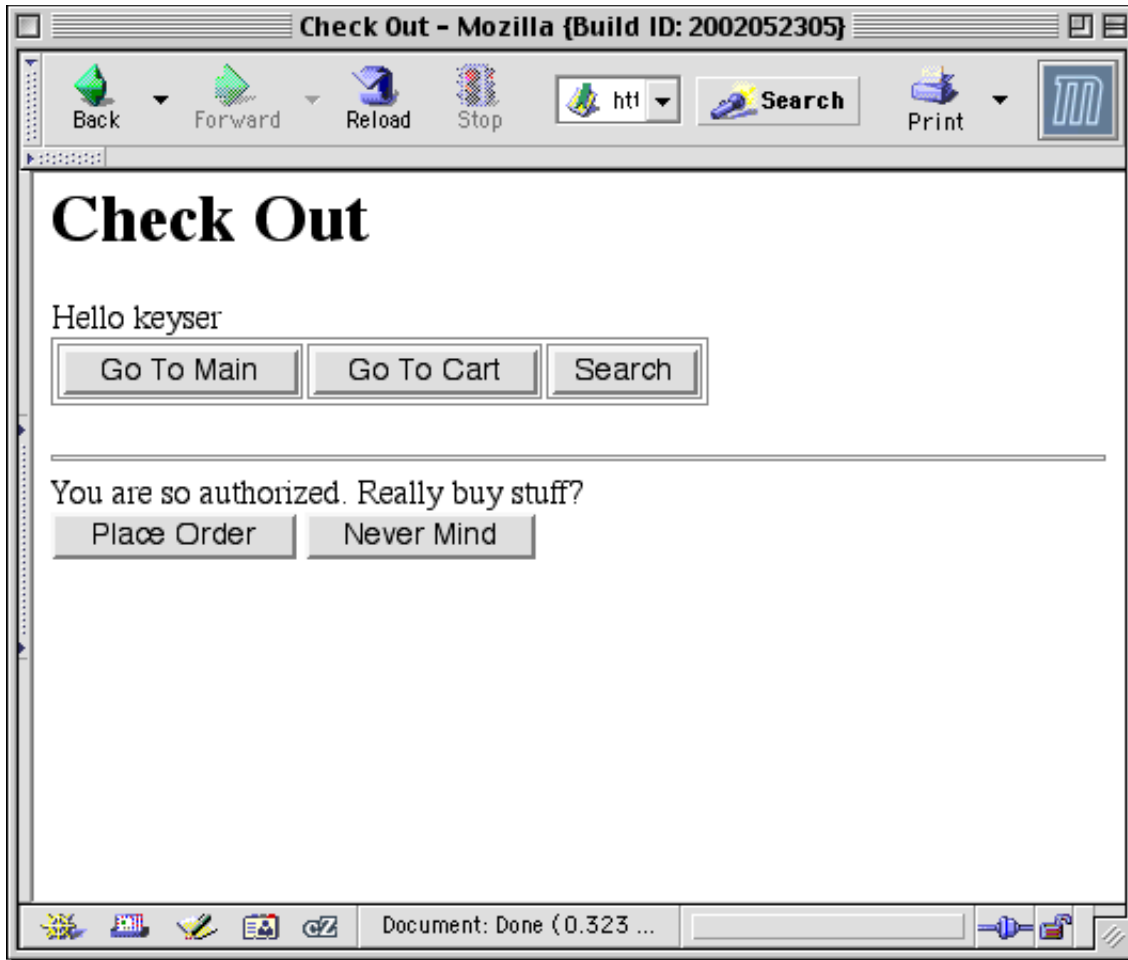
If the user checks the "Remember Name" checkbox, when submitting c2 with the Login button, then a persistent cookie should be set so that...

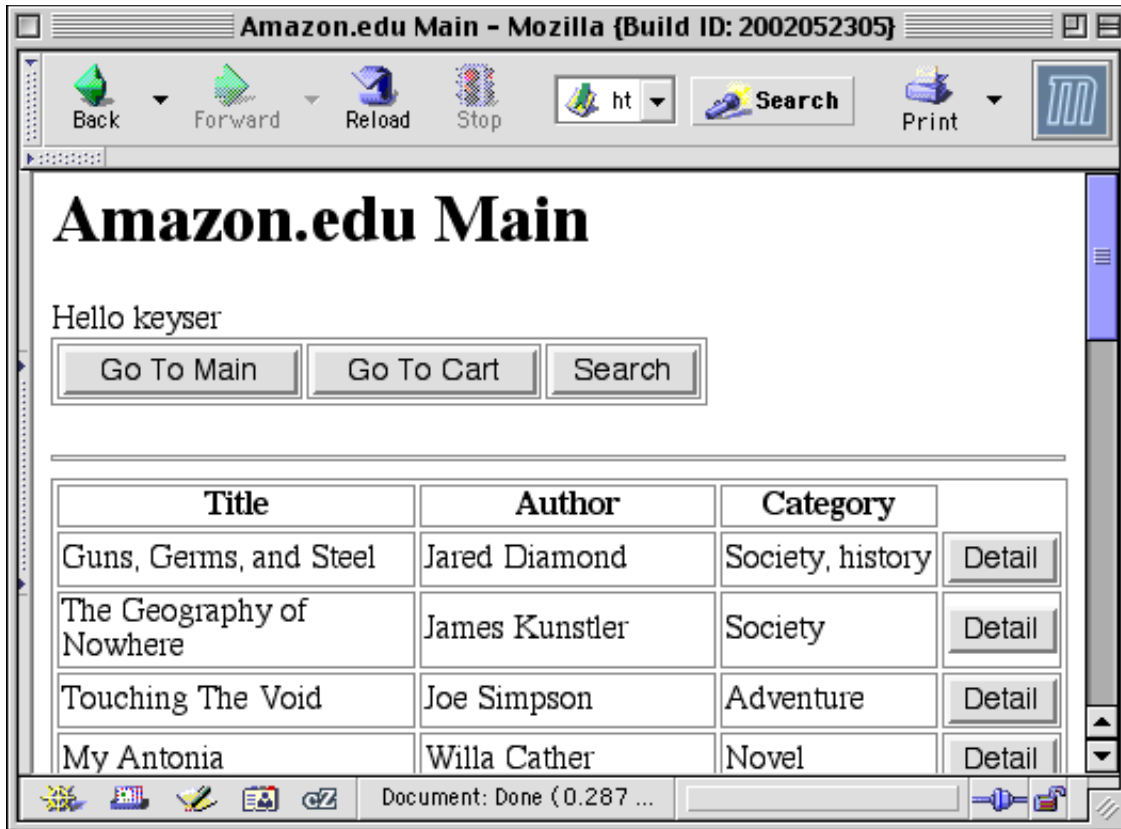
1. The top of every page includes a "Hello <name>" greeting below the buttons
2. The next time the user comes to the c1 page, the name field should already be filled in.

These effects should continue to work, even if the user restarts their browser and re-connects to Amazon.edu. If the "Remember name" check box is not checked when the login button is used with a correct name and password, then the cookie should be cleared, so there is not a greeting and the name is not pre-filled in the form.

Here's an example sequence where Keyser Soze logs in, but then hits the Go To Main button -- notice the greeting at the top of the pages...







XML Passwords

To be a properly hip web-app, we'll store the username/password data in a simple users.xml file...

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user name="foo" password="foo" />
  <user name="bar" password="bar" />
  <user name="keyser" password="keyser" />
  <user name="robert" password="bob" />
</users>
```

Create a XMLPassword class that uses the SAX parser to read users.xml file into its ivars and responds to a boolean checkPassword(name,pass) message. It's acceptable for the XMLPassword to be implemented very simply -- store the names in one ArrayList and the passwords in another, and use simple String.equals() to compare strings. The servlet should create and init the XMLPassword the first time it needs to check a password. (Feel free to use the lecture XMLDotReader as a starting point for XMLPassword.)

Part C -- JavaScript

For the final feature, we will use JavaScript to implement a "search" feature that runs entirely on the client side. Add a "Fancy Search" button to the standard form, which leads to the Fancy Search page...



It does not look all that different from the search/results style from our earlier CGI work, however under the hood, fancy search works in a completely different way. Fancy search is implemented entirely with JavaScript. The JavaScript techniques shown here should work with the modern, standards compliant browsers -- the open-source Mozilla browser or IE 6. They will partially work on earlier browsers.

When the page is generated, it includes...

- JavaScript code that builds an array containing the title, author, and category strings for all the books. There is only one

request/response to make the page, after which the page has all the book data.

- A JavaScript `generate()` function that looks at the current value of the input field, iterates through the array, and generates the `<tr>..</tr>` for each row that includes the target string (not case sensitive). For the main page, the table generation was done in Java. Here, the table generation is done in JavaScript running on the client side.
- The little search form and input field (show below). The form should include an "onsubmit" binding that calls `generate()` and returns false (since we don't actually want the form to submit). The text field should include an "onkeyup" binding that calls `generate()` -- in this way, we generate the correct table keystroke by keystroke. The text field may also include an "onchange" binding that calls `generate()`, since `onkeyup` is not implemented by all browsers. `Onchange` should fire when the user hits return.
- A little `<div id="result"> </div>` tag which is ready to receive the table.

Array Definition

```
<script language='JavaScript'>
lines = new Array();
lines.push(["Title", "Author", "Category",]);
lines.push(["Guns, Germs, and Steel", "Jared Diamond", "Society, history",]);
...
lines.push(["The Sweet Hereafter", "Russel Banks", "Novel",]);
</script>
```

Form

```
<form name=frm onsubmit="generate(); return false;">
Search:<input type=text name=target onkeyup="generate();" ></form>
```

```
<script language="JavaScript" >

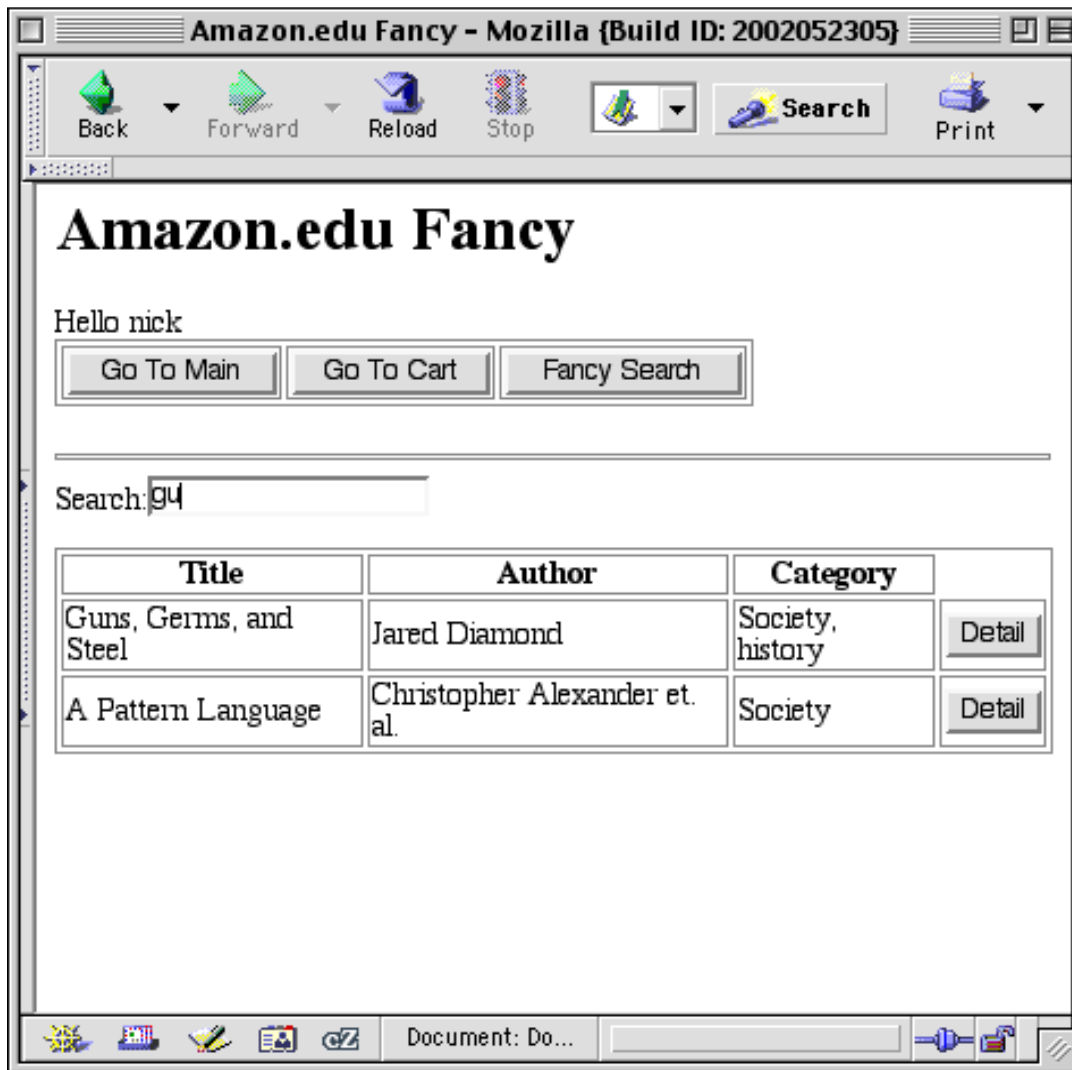
    // YOUR CODE HERE

</script>
```

Results

```
<div id="result"></div>
</body>
</html>
```

Each letter typed in the text field, calls generate() on the client side, which re-generates the table to include only the matching rows...



Servlet vs. JSP

The `<script>` section that defines the array is most easily written out by the main servlet with `println()`. However, the `<script>` section that includes the form and the JavaScript functions is very inconvenient to write with `println()`. Therefore, write the form and the JavaScript functions in separate `"/fancy.html"` that the servlet includes in its response.

Java vs. JavaScript

Amazon depends on a basic transformation of plain data into HTML table data...

```
Guns, Germs, and Steel    -->    <td>Guns, Germs, and Steel</td>
```

For Part A, you need to write this code in Java to generate the main page. For the JavaScript section, there are two possible strategies, and you may choose whichever you prefer...

- Generate the JavaScript arrays to contain plain data. Write JavaScript code that looks at the plain data and generates the HTML on the fly. This has the advantage that the arrays are compact.
- On the servlet side, generate the JavaScript arrays with the HTML formatting already done. This makes the JavaScript code a little simpler, and allows you to re-use the Java code from Part A. The downside is that the JavaScript arrays, which must be sent on the HTTP response, are now bigger