# Building Web Applications

Jerry Cain
CS 106AX
November 29, 2023

# Building Web Applications

- Most web applications bias toward a ***thin-client*** architecture, which means most of the computationally intense program logic is managed by the server. The client, conversely, requires a minimal amount of processing power and typically manages the user experience and virtually nothing else.
- Programs coded to such an architecture are organized across several different files:
  - HTML files housing the smallest amount of markup needed to structure the web pages, alongside CSS files supplying rules to inform the browser how to style them.
  - A collection of server-side endpoints that are operationally a library of functions invoked using networking and URLs.
  - JavaScript code that knows when to invoke these endpoints (using **`AsyncRequest`**) and update the DOM in response.

## Building Web Applications

- One of many important aspects of the architecture is to ensure the client and server are consistent—that is, the browser presentation of information is in sync with the information stored on the server.
- To that end, the server typically defines a collection of server-side endpoints that comprise an API. In web programming, the API is comprised of a collection of URLs that can be used to make remotely executing functions ***appear*** to run locally.
- When we make API calls like **`/factor.py?number=14728`**, we are invoking a remotely implemented function called **`factor.py`**. **`factor.py`** expects its arguments to be expressed within the query string (here, one argument called **`number`**). The factorization is returned via the response payload.

# Example: Persistent To-Do List

- We’ll revisit the to-do list application from last week and implement one key difference: we’ll require a server to store a persistent copy of the list that can be syndicated to every browser that ever connects to it.
- In order to fully realize the application, we need to define the structure and behavior of a small number of endpoints allowing us to synchronize application state between client and server. Those endpoints are:
  - **`GET /scripts/getListItems.py`**, which returns a JSON object of item-id/item-text pairs.
  - **`POST /scripts/addListItem.py`**, which posts the text of a new item so it’s shared with and stored by the server.
  - **`POST /scripts/removeListItems.py`**, which posts a JSON list of item-ids that the server should delete.

# Example: Persistent To-Do List

File: **`todo.html`**

```html
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8">
      <title>To-Do List</title>
      <script src="async.js" type="text/javascript"></script
      <script src="todo.js" type="text/javascript"></script>
   </head>
   <body>
      Today's To-Do List:
      <ul id="to-do-list">
         <!-- This is where list items will be placed. -->
      </ul>
      New item: <input id="item-text" size="60"/>
      <button id="add-item" type="button">Add To List</button>
      <button id="clear-all-items" type="button">Clear</button>
   </body>
</html>
```

Very often the HTML is small and only includes elements that are always visible—e.g., the text input, the two buttons, and a skeletal unordered list. We rely on JavaScript and **`AsyncRequest`** to pull in list items, which potentially vary with each page load.

## Example: Persistent To-Do List

File: **`/scripts/getListItems.py`**

```
def extractItems(name): # used by all three endpoints
    with open(name) as infile:
        return json.loads(infile.read())

def publishPayloadAsJSON(response): # used by all three endpoints
    responsePayload = json.dumps(response)
    print("Content-Length: " + str(len(responsePayload)))
    print("Content-Type: application/json")
    print()
    print(responsePayload)

info = extractItems("items.json")
publishPayloadAsJSON(info["items"])
```

This is the server-side endpoint that will be called by the onload **`AsyncRequest`**. The response payload will look like this:

```
{
   4: "Pick up dry cleaning.",
   5: "Call Wells Fargo about mortgage rate.",
   6: "Write last CS106AX final exam question.",
   8: "Schedule dinner with Avi and SLs."
}
```

## Example: Persistent To-Do List

File: `todo.js (part 1)`

```
function BootstrapToDoList() {
   let ul = document.getElementById("to-do-list");
   let input = document.getElementById("item-text");
   // other variables omitted for brevity

   AsyncRequest("/scripts/getListItems.py")
     .setSuccessHandler(populateInitialList)
     .send(); # notation provided to daisy chain method calls

   function populateInitialList(response) {
      while (ul.lastChild !== null) ul.removeChild(ul.lastChild);
      let items = JSON.parse(response.getPayload());
      for (let key in items) appendListItem(key, items[key]);
   } // implementation of appendListItem on next slide
}

document.addEventListener("DOMContentLoaded", BootstrapToDoList);
```

The above JavaScript is a subset of the full ***controller***, but just enough to be clear how the to-do list is populated with the current set of items as stored server-side.  This programming model works to ensure the web page and server can always be in sync.

# Example: Persistent To-Do List

File: **`todo.js (part 2)`**

```
    function appendListItem(itemid, text) {
        let li = document.createElement("li");
        let tn = document.createTextNode(text);
        li.setAttribute("id", "item-" + itemid)
        li.setAttribute("data-id", itemid);
        li.addEventListener("dblclick", onItemDoubleClick);
        li.appendChild(tn);
        ul.appendChild(li);
    }
```

The implementation of **`appendListItem`** should be familiar from prior versions of our to-do list application. We create a text node around the supplied text and embed it within a new **`<li>`** node. This new **`<li>`** node is appended to the running list of **`<li>`** nodes already hanging from **`ul`**. The key differences? We add an **`id`** attribute so the node can be programmatically discovered by **`document.getElementById`**, and we add a **`data-id`** attribute that stores the raw list item identifier as it's stored server-side.

## Example: Persistent To-Do List

File: **`todo.js (part 3)`**

```
function onAddClick(e) { // e argument is ignored
   let text = input.value.trim();
   input.value = "";
   if (text.trim() == 0) return;

   AsyncRequest("/scripts/addListItem.py")
     .setMethod("POST") # defaults to "GET", so override
     .setPayload(JSON.stringify(text))
     .setSuccessHandler(addNewListItem)
     .send();
}

function addNewListItem(response) {
   let payload = JSON.parse(response.getPayload());
   appendListItem(payload.id, payload.item);
}
```

**`onAddClick`** is invoked whenever the user clicks the **`Add To List`** button. We want this new item to persist server-side, so we issue a **`"POST"`** request via an **`AsyncRequest`**, and install a response callback that inserts a new **`<li>`** item to the DOM. This model ensures the server and the client agree on the state of the list.

## Example: Persistent To-Do List

File: **`/scripts/addListItem.py`**

```
requestPayload = json.loads(extractPayload())
info = extractItems("items.json")
id = info["id"]
newItem = {"id": id, "item": requestPayload }
info["id"] += 1
info["items"][id] = requestPayload
saveAllItems("items.json", info)
publishPayloadAsJSON(newItem)
```

The **`addListItem.py`** endpoint is invoked by the **`AsyncRequest`** outlined on the previous slide. As the client is uploading new information , the request method is **`"POST"`** and the request includes a payload (returned by a library function called **`extractPayload`**). The payload is a string constant, which is valid JSON provided it includes the double quotes. Our endpoint associates the next available id with the uploaded text, updates the internal store to include the association, and publishes a response whose payload includes the original text along with the assigned id.

## Example: Persistent To-Do List

File: **`todo.js (part 4)`**

```
    function onClearClick(e) { // e argument is ignored
        let itemIDs = [];
        for (let i = 0; i < ul.childNodes.length; i++) {
            itemIDs.push(ul.childNodes[i].getAttribute("data-id"));
        }
        AsyncRequest("/scripts/removeListItems.py")
            .setMethod("POST")
            .setPayload(JSON.stringify(itemIDs))
            .setSuccessHandler(removeListItems)
            .send();
    }
```

**`onClearClick`** is invoked whenever the user clicks the **`Clear`** button. It collects all of the text item ids into an array, and then posts that array to the third of our three server-side endpoints, **`/scripts/removeListItems.py`**. We install **`removeListItems`** to be invoked once the server responds with information about which items were successfully deleted (generally all of them) and which ones were not.

## Example: Persistent To-Do List

File: **`todo.js (part 5)`**

```
    function removeListItems(response) {
       let itemsToDelete = JSON.parse(response.getPayload());
       for (let key in itemsToDelete) {
          let li = document.getElementById("item-" + key);
          li.parentNode.removeChild(li);
       }
    }
```

The above is invoked on the **`removeListItems.py`** endpoint responds with a payload that is structured as follows:

```
{
   4: true,
   5: true,
   6: true,
   8: true
}
```

The **`for`** loop instructs each of the relevant **`<li>`** nodes to remove itself from its **`<ul>`** parent.

## Example: Persistent To-Do List

File: **`/scripts/removeListItems.py`**

```
itemsToDelete = json.loads(extractPayload())
info = extractItems("items.json")
results = {};
for itemID in itemsToDelete:
   results[itemID] = itemID in info["items"]
   if results[itemID]:
      del info["items"][itemID]
saveAllItems("items.json", info)
publishPayloadAsJSON(results)
```

The above endpoint expects the request payload to be structured as an array of item ids of those items that should be removed from the data store.  It does precisely that for each id, allowing for the possibility that an id is no longer present, because a second client deleted the item and the first is just out of sync.  The script responds with information about what items were truly deleted and which ones weren’t.

# Example: Persistent To-Do List

File: `todo.js (part 6)`

```
function onItemDoubleClick(e) {
   let itemID = e.target.getAttribute("data-id");
   let itemIDs = [itemID]
   AsyncRequest("/scripts/removeListItems.py")
      .setMethod("POST")
      .setPayload(JSON.stringify(itemIDs))
      .setSuccessHandler(removeListItems)
      .send();
   }
}
```

The **`onItemDoubleClick`** handler uses the same endpoint **`onClearClick`** does. In this case, we extract the item id right out of the element that triggered the callback, embed that item in an array as **`/scripts/removeListItems.py`** expects, and post the request to the server. Note we're able to install the same success handler used by **`onClearClick`**, which was coded specifically to handle the both the remove-all and remove-one user interactions.

The End