

ECE 448/528

Application Software Design

Lecture 19. JavaScript and DOM – Part II

Spring 2025

Won-Jae Yi, Ph.D.

Department of Electrical and Computer Engineering
Illinois Institute of Technology

Event-Driven Web UI Design

Why event-driven?

- Complex applications need to interact with users via UI and communicate with servers via networking at the same time.
 - In general, there are interactions between the application and the external environment (user and network).
 - Need a consistent mechanism to handle both.
- Event-driven applications
 - Process external events: user inputs, responses from servers.
 - Cause external events to happen: update UI elements, send requests to servers.

Event-Driven Web UI

- JavaScript code, when used on a web page, is organized into pieces that react to certain events.
 - e.g., `throw_dice` is called when the button is clicked.
- To avoid complexity in synchronization, all such event handlers in JavaScript are called from 1 SINGLE thread.
 - Greatly simplifies the reasonings, especially when you need to handle responses from the servers that can arrive at any moment.
- As a byproduct, those event handlers must be cooperative.
 - Each event handler should finish as soon as possible.
 - If an event handler takes too long to finish, other event handlers must wait – the web page will become unresponsive.
- Modern JavaScript introduces `Promise` to break lengthy handlers into smaller ones.
 - A `Promise` is an object that may produce a single value to be used sometime later in the future
 - Essentially, `Promise`'s create additional internal events that are handled at a later time (e.g., to display data retrieved from the server)
 - Now, why `Promise` was created in JavaScript? Why we need this?

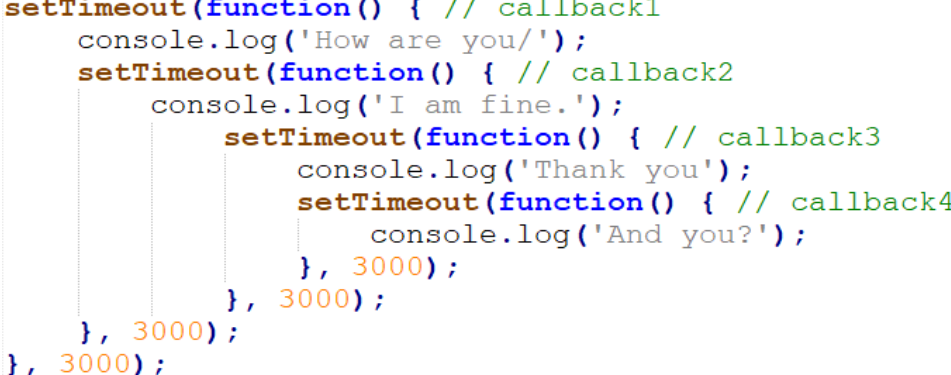
Why use Promise?

- For Async processing, `callback` was used. For example, `function()` will be executed 3 seconds after when the timer is up

```
setTimeout(function() { // callback
  console.log('Bye');
}, 3000);
```

- Now, a `callback hell` can occur if using this async logic with many `callbacks`...

```
setTimeout(function() { // callback1
  console.log('How are you/');
  setTimeout(function() { // callback2
    console.log('I am fine.');
```



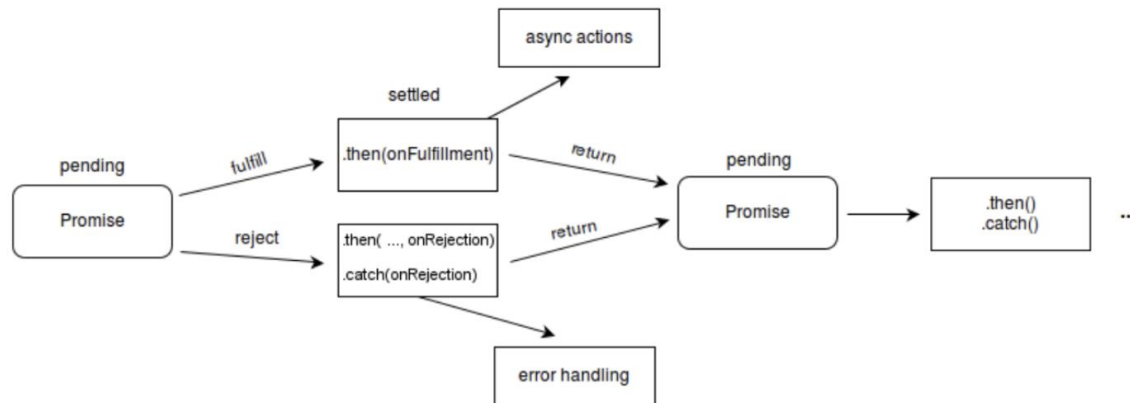
```
    setTimeout(function() { // callback3
      console.log('Thank you');
```

```
      setTimeout(function() { // callback4
        console.log('And you?');
```

```
      }, 3000);
    }, 3000);
  }, 3000);
}, 3000);
```

Chained Promises

- `Promise` has three states
 - Fulfilled: desired action is resolved/successfully completed
 - Rejected: desired action is rejected due to operation fail
 - Pending: desired action is neither fulfilled/rejected
- `Promise.then()`
 - callback functions can be inserted as callback queue
 - can take up to two parameters (*resolved, rejected*)
 - return value will be another `promise`
- `Promise.catch()`
- `Promise.finally()`



A Complex Web Application

Web UI for Groups

- Allow users to manipulate groups via a web UI.
 - Need to communicate with our RESTful server backend.
- User stories
 - i. As an end-user, I want to see the names and members of all groups sorted by their names, so that I can check all of them in the same place.
 - ii. As an end-user, I want to input the name and the members to add or replace a group, so that I can modify groups.
 - iii. Additional ones?

Identifying Events I

- User story I: ... I want to see the names and members of all groups sorted by their names ...
 - When do you want to see them?
 - When the web page is refreshed.
 - But the names and the members are not available at this moment – they are on the RESTful server.
 - You won't be able to see them right at the moment – just think about the extreme case where the RESTful server is not responsive at the moment.
- We need to handle two events.
 1. Web page is loaded: we need to send the RESTful request to get all groups.
 2. Response to RESTful request arrives: we need to update the UI elements to show groups.
- In most cases, the time between these two events is so small that most people won't notice.
 - Though as developers we would need to understand there are two events, one happens after another.

Implementing User Story I

```
<div id="groups">
  Loading groups ...
</div>
...
<script>
function get_groups() {
  ... // request info from RESTful server
}
function show_groups(groups) {
  ... // update the "groups" div
}
get_groups(); // run when page is loaded
</script>
```

- Use our Spring Boot application built in Lecture 16/17.
 - In `public/index.html` under branch `lec19-groups`
- Define two functions for the RESTful request/response.
- Send RESTful request when the page is loading.

fetch

```
function get_groups() {  
  fetch("api/groups")  
    .then(rsp => rsp.json())  
    .then(groups => show_groups(groups))  
    .catch(err => console.debug("Groups: error", err));  
}
```

- The `fetch` function is used to send RESTful requests.
 - `fetch(resource)` or `fetch(resource, options)`
 - By default, the GET method is used so you just need to provide the path.
 - Options can include GET, POST, DELETE, ...
 - The path (resource) is relative to the file containing the script.
- `Fetch` returns a `Promise`: you'll need to provide event handlers to process the outcomes (success or failure).
 - By calling the method `then` (success/resolved) or `catch` (failure/error).
 - These names somewhat allow you to read those complex function calls in plain English.

Lambda Functions

```
function get_groups() {  
  fetch("api/groups")  
    .then(rsp => rsp.json())  
    .then(groups => show_groups(groups))  
    .catch(err => console.debug("Groups: error", err));  
}
```

- The event handlers should be functions in JavaScript.
 - It is convenient to use lambda functions.
- Modern JavaScript supports arrow function expressions to define lambda functions.
 - Syntax is very similar to that of Java: `params => function body`
- Be careful: the capture rules are different than Java.
 - We will cover them later as needed.

Handling Response and Failures

```
function get_groups() {  
  fetch("api/groups")  
    .then(rsp => rsp.json())  
    .then(groups => show_groups(groups))  
    .catch(err => console.debug("Groups: error", err));  
}
```

- Lambda in the first **then** handles the response from the RESTful request.
 - It will convert the response into JSON.
 - This may take a lot of time for large responses, so JavaScript requires it to be done via another **Promise** – that's why we have a second **then**.
- Lambda in the second then handles the outcome of the conversion.
 - It updates the UI elements.
- Any errors are handled by **catch**.
 - Including syntax errors.

UI Updates

```
function show_groups(groups) {  
    groups.sort((l, r) => l.name.localeCompare(r.name));  
  
    var html = "";  
    for (var group of groups) {  
        html += "<div>" + group.name + ": [" + group.members + "]</div>";  
    }  
  
    document.getElementById("groups").innerHTML = (groups.length == 0)?  
        "There is no group.": html;  
}
```

- Use lambda function to define how elements are compared for sorting.
- Use **for** loop to form the whole list of groups.
 - Modern JavaScript supports **for of** loops to iterate through array elements.

Identifying Events II

- User story II: ... I want to input the name and the members in order to add or replace a group ...
 - Should we update our display of all groups?
 - Yes, we should. But when?
 - Option 1: after the user clicks the button to add/replace.
 - Option 2: we may initiate another RESTful request to get all groups after adding the group.
 - Choose option 2 as we want to make sure the server adds/replaces the group correctly.
- We need to handle three events.
 - Button is clicked: we need to send the RESTful request to create the group.
 - Response to create group request when arrives: we need to send the RESTful request to get all groups.
 - Response to get all groups' requests when arrives: we need to update the UI elements to show groups.

fetch by POST

```
function create_group() {  
  var name= document.getElementById('group_name').value;  
  var members=document.getElementById('group_members').value.split(',');  
  console.info("Groups: add "+name+" with members "+members);  
  
  var post_req = {  
    method: "POST",  
    headers: {"Content-Type": "application/json"},  
    body: JSON.stringify(members)  
  };  
  fetch("api/groups/"+name, post_req)  
    .then(rsp => get_groups())  
    .catch(err => console.error("Groups:", err));  
}
```

- `fetch` takes a second parameter to specify the details of the request.
- We don't care what that POST returns – just call `get_groups` to initiate another RESTful request.

Additional User Stories

- Provide buttons to delete a group.
- Click on an existing group to fill in the inputs so that groups can be replaced easily.
 - Click an existing group to modify it in place.

Discussions

- As more features are added, the web application will become more complicated.
 - Need to handle many more events.
 - Need to identify UI elements to be updated for each event.
 - Need to assemble more strings to update the HTML DOM.
- We need a more systematic approach!
 - Better organization of events and event handlers.