Web Application Development for Industrial Engineers

การพัฒนาเวปแอปพลิเคชันสำหรับวิศวกรอุตสาหการ

Asynchronous Programming

Motivation

- Normally, a given program's code runs straight along, with only one thing happening at once.
- If a function relies on the result of another function, it has to wait for the other function to finish and return.
    - And until that happens, the entire program is essentially stopped from the perspective of the user.
- This is **synchronous programming**.

Motivation

- synchronicity can lead to frustrating experience for a user.
- Also it might not be a good use of computer processing power.
  - Computers have multiple processor cores available.
  - We should let the other task get processed on another processor core and let us know when it's done.
- This is the basis of **asynchronous programming**.

Environment

- It is up to the programming environment you are using (web browsers, in the case of web development) to provide you with APIs that allow you to run such tasks asynchronously.

Synchronous JavaScript

- A lot of the functionality we have larned are synchronous.

```html
<button>Click me</button>
```

```javascript
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  alert('You clicked me!');

  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

Blocking code

- When a web app runs in a browser and it executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen.
- This is called **blocking**.

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  // Expensive calculations
  let myDate;
  for (let i = 0; i < 10000000; i++) {
    let date = new Date();
    myDate = date;
  }
  console.log(myDate);
  // -->

  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

Side note: thread

- JavaScript, generally speaking, is **single-threaded**.
- A thread is basically a single process that a program can use to complete tasks. Each thread can only do a single task at once.
- Even with multiple cores, you could only get it to run tasks on a single thread, called the **main thread**.

What happened?

- The code above contains all synchronous tasks.
- Tasks are executed sequentially.

```
Main Thread: Task A --> Task B --> Task C --> Task D ...
```

Asynchronous javascript

- `setTimeout`
- `setInterval`

setTimeout

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  // Asynchronous task
  setTimeout(() => alert('I waited.'), 3000);
  //
  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

What happened?

- `setTimeout` cause the "task" (callback function) be to asynchronous.

```
Main Thread: Fire Handler --(Skip alert)--> Show paragraph --(After 3 seconds)--> Show alert.
```

Passing extra argument

```
setTimeout(() => alert('I waited'), 3000);

// Or

setTimeout(alert, 3000, 'I waited');
```

What is the value of `x` from `alert`?

```
let x = 10;
setTimeout(alert, 3000, x);
x = 20;
console.log(x);
```

```
let x = 10;
setTimeout(() => alert(x), 3000);
x = 20;
console.log(x);
```

`setInterval`

```
<p id="clock"></p>
<button>Click me</button>
```

```
const clock = document.querySelector('#clock');
// Asynchronous task
setInterval(() => {
  const date = new Date();
  clock.textContent = date.toLocaleTimeString();
}, 1000);

const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  let pElem = document.createElement('p');
  pElem.textContent = 'This is a newly-added paragraph.';
  document.body.appendChild(pElem);
});
```

- If `setInterval` did not create a asynchronous task, the button would not work!

clearInterval

```
<p id="clock"></p>
<button id="start">Start Clock</button>
<button id="stop">Stop Clock</button>
```

```
const clock = document.querySelector('#clock');
const btnStart = document.querySelector('#start');
const btnStop = document.querySelector('#stop');

let timer;
btnStart.addEventListener('click', () => {
  // Asynchronous task
  timer = setInterval(() => {
    const date = new Date();
    clock.textContent = date.toLocaleTimeString();
  }, 1000);
});

btnStop.addEventListener('click', () => {
  clearInterval(timer);
});
```

Async callbacks

- Async callbacks are functions that are specified as arguments when calling a function which will start executing code in the background.
- When the background code finishes running, it calls the callback function to let you know the work is done, or to let you know that something of interest has happened.
- Using callbacks is slightly old-fashioned now, but you'll still see them in use in a number of lder-but-still-commonly-used APIs.

Asynchronous programming with Promises

`fetch`

- The Fetch API is a modern interface that allows you to make HTTP requests to servers from web browsers.
- The `fetch()` method is available in the global scope that instructs the web browsers to send a request to a URL.
- Let's `fetch` information from Star Wars API

`fetch`

```
const result = fetch('https://swapi.dev/api/people/1');

console.log(result); // Promise { <state>: "pending" }
```

- The `fetch()` method returns a `Promise`.
- But what is a **Promise**?

Promise (def. 1)

- A `Promise` allows you to defer further actions until after a previous action has completed, or respond to its failure.
- This is useful for setting up a sequence of async operations to work correctly.

Promise (def. 2)

- A `Promise` is an object that represents an intermediate state of an operation — in effect, a promise that a result of some kind will be returned at some point in the future.
- There is no guarantee of exactly when the operation will complete and the result will be returned but there is a guarantee that when the **result is available**, or **the promise fails**.
- You can then write the code that will be executed in order to do something else with a successful result, or to gracefully handle a failure case.

Promise (def. 3)

A `Promise` is in one of these states:

- `pending`: initial state, neither fulfilled nor rejected.
- `fulfilled`: meaning that the operation was completed successfully.
- `rejected`: meaning that the operation failed.

Exploring the states of a Promise

(Use firefox console)

```
const a = new Promise(() => {}); //Promise { <state>: "pending" }s

const b = Promise.resolve(); // Promise { <state>: "fulfilled", <value>: undefined }

const b = Promise.resolve('I waited.'); // Promise { <state>: "fulfilled", <value>: "I waited." }

const c = Promise.reject(); // Promise { <state>: "rejected", <reason>: undefined }

const c = Promise.reject('Wrong'); // Promise { <state>: "rejected", <reason>: "Wrong" }
```

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('foo'); //or reject('foo')
  }, 5000);
});
```

- You can keep typing `myPromise` in the Firefox console to see the state changed.

Responding to a Promise

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject('foo'); //or reject('foo')
  }, 300);
});
```

```
myPromise.then((msg) => console.log(msg)).catch((err) => console.log(err));
```

- You can use `quokka` to see which block get executed.

Returning from `then`

```
myPromise
  .then((msg) => {
    console.log(msg);
    return 'bar';
  })
  .then((msg) => console.log(msg));
```

Returning a promise

```
myPromise
  .then((msg) => {
    console.log(msg);
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve('bar');
      }, 3000);
    });
  })
  .then((msg) => console.log(msg));
```

(Run in a web browser to see the log timing.)

Promise factory

```
function wait(msg, time) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(msg), time);
  });
}
```

```
wait('First', 1000)
  .then((msg) => {
    console.log(msg);
    return wait('Second', 2000);
  })
  .then((msg) => {
    console.log(msg);
    return wait('Third', 3000);
  })
  .then((msg) => {
    console.log(msg);
    return wait('Fourth', 4000);
  })
  .then((msg) => {
    console.log(msg);
  });
```

Back to `fetch`

```
<h1>Star Wars</h1>
<p id="p1"></p>
<p id="p2"></p>
```

Single Promise

```
const p1 = document.querySelector('#p1');

fetch('https://swapi.dev/api/people/1')
  .then((res) => res.json())
  .then((data) => {
    // console.log(JSON.stringify(data));
    p1.textContent = JSON.stringify(data);
  });
```

With error handling

```
const p1 = document.querySelector('#p1');

fetch('https://swapi.dev/api/people/random_stuff')
  .then((res) => res.json())
  .then((data) => {
    p1.textContent = JSON.stringify(data);
  })
  .catch((err) => {
    p1.textContent = err;
  });
```

Multiple dependent requests

```
fetch('https://swapi.dev/api/people/1')
  .then((res) => res.json())
  .then((data) => {
    p1.textContent = JSON.stringify(data);
    return fetch(data.homeworld); // "https://swapi.dev/api/planets/1/"
  })
  .then((res) => res.json())
  .then((data) => {
    // console.log(data);
    p2.textContent = JSON.stringify(data);
  });
```

Multiple independent requests

- The `Promise.all()` method takes an iterable of promises as an input, and returns a single `Promise` that resolves to an array of the results of the input promises.

```
const ul = document.querySelector('ul');

fetch1 = fetch('https://swapi.dev/api/people/1');
fetch2 = fetch('https://swapi.dev/api/people/2');
fetch3 = fetch('https://swapi.dev/api/people/3');
fetch4 = fetch('https://swapi.dev/api/people/4');

Promise.all([fetch1, fetch2, fetch3, fetch4])
  .then((responses) => {
    return responses.map((response) => response.json());
  })
  .then((array) => {
    array.forEach((el) => {
      el.then((data) => {
        const li = document.createElement('li');
        li.textContent = JSON.stringify(data);
        ul.appendChild(li);
      });
    });
  });
```

`async` and `await`

- An `async` function is a function declared with the `async` keyword, and the `await` keyword is permitted.
- The `async` and `await` keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

```
const p1 = document.querySelector('#p1');

async function getData(url) {
  const res = await fetch(url);
  const data = await res.json();
  p1.textContent = JSON.stringify(data);
}

getData('https://swapi.dev/api/people/1');
```

Error handling

```
const p1 = document.querySelector('#p1');

async function getData(url) {
  try {
    const res = await fetch(url);
    const data = await res.json();
    p1.textContent = JSON.stringify(data);
  } catch (err) {
    p1.textContent = JSON.stringify(err);
  }
}

getData('https://swapi.dev/api/people/random_staff');
```

Multiple independent requests

```
const ul = document.querySelector('ul');

const urls = [
  'https://swapi.dev/api/people/1',
  'https://swapi.dev/api/people/2',
  'https://swapi.dev/api/people/3',
  'https://swapi.dev/api/people/4',
];

async function getData(url) {
  const res = await fetch(url);
  const data = await res.json();
  const li = document.createElement('li');
  li.textContent = JSON.stringify(data);
  ul.appendChild(li);
}

urls.forEach((url) => getData(url));
```