

Bonus: ES8 Async & Await

Shan-Hung Wu & DataLab
CS, NTHU

Outline

- ES6 Promises
- ES8 Async & Await
- Pitfalls

```
// in method1()
const p = new Promise((resolve, reject) => {
  ... // do asynchronous job here
  if (success) resolve(data);
  else reject(err);
});
return p;
```

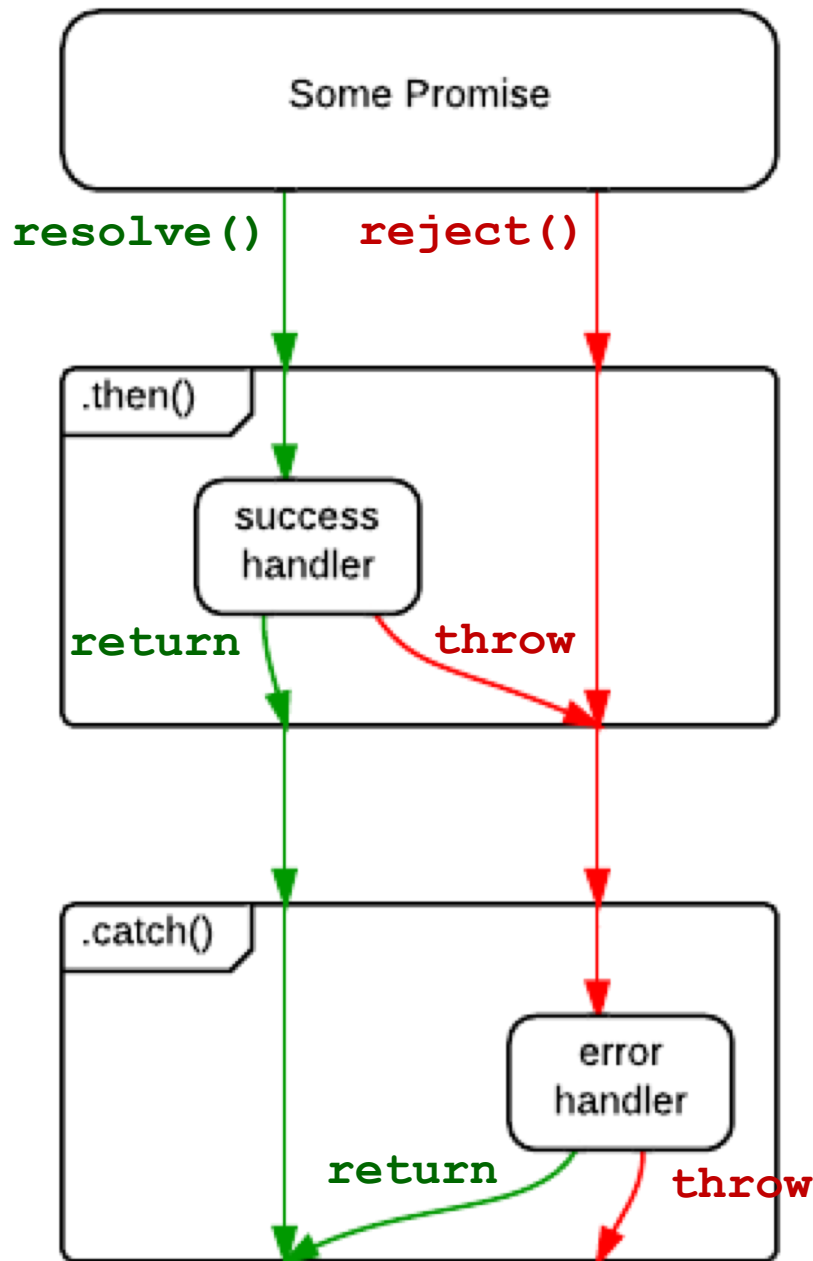
```
// in method2(p)
const p2 = p.then(data => {
  ... // process data
  return data2
}); // always returns a new Promise
return p2;
```

```
// in method3(p2)
p2.then(data2 => {
  ... process data2
}).catch(err => {
  ... // handle err
}); // always returns a new Promise
```

ES6 Promise

- A value available *in the future*
- Separation of concerns
 - Handlers can be written in different places
- Use arrow func for this

Execution Flow



- Chain `then` and/or `catch` as long as you like
- Reject mode:
 - `throw new Error()`
- Resolve mode:
 - `return`

Axios and AJAX Requests

```
const axios = require('axios');

// GET request
axios.get('...url...').then(res => {
  res.status // HTTP response code (e.g., 200, 401)
  res.data   // object parsed from HTTP response body
  res.headers // HTTP response headers
}).catch(err => {
  console.log(err.response.status);
});

// POST request
axios.post('...url...', {
  ... // request body
}).then(...).catch(...);
```

- Requests can be [canceled](#)

Outline

- ES6 Promises
- ES8 Async & Await
- Pitfalls

ES8 Async & Await

- Goal: to make asynchronous code looks more consistent with synchronous code
- Now, supported by major browsers and Node.JS v7.6+

Example

```
// ES6 Promise
function getFirstUser() {
  return getUsers().then(users => users[0].name)
    .catch(err => ({
      name: 'default user'
    }));
}
```

```
// ES7 Async/Await
async function getFirstUser() {
  try {
    // line blocked until promise
    // resolved/rejected.
    let users = await getUsers();
    return users[0].name;
  } catch (err) {
    return {
      name: 'default user'
    };
  }
}
```

- An *async* function returns a promise
- *Await* on a promise until value available
- *Try/catch* for resolve/reject

AJAX, the Async/Await Style

```
const axios = require('axios');

async function getFirstUser() {
  try {
    let users = await axios.get('...url...');
    return users[0].name;
  } catch (err) {
    console.log(err.response.status);
    return {
      name: 'default user'
    };
  }
}
```

Outline

- ES6 Promises
- ES8 Async & Await
- **Pitfalls**

Pitfall #1: Mixing Callbacks/Promises

- Choose one style, and stay consistent in your *entire* project
 - Callbacks
 - ES6 Promise
 - ES8 Async/await
- Increases readability of your code

Pitfall #2: `await` Not in `async` Function

- You can use `await` at top level
- But when using `await` is in a function, the function must be `async`

```
await axios.get('...url...'); // OK
```

```
function getUser(id) {  
  return await axios.get('...url...'); // error  
}
```

```
async function getUser(id) {  
  return await axios.get('...url...'); // OK  
}
```

Pitfall #3: `Array.forEach()`

```
const axios = require('axios');

async function getUsers(ids) {
  let users = [];
  try {
    ids.forEach(async id => {
      let user = await axios.get('...url...');
      users.push(user);
    });
  } catch (err) {...}
  return users;
}
const vips = await getUsers(...);
```

- **Empty users returned immediately**
 - Elements added later

Solution: Use `for` Instead

```
const axios = require('axios');

async function getUsers(ids) {
  let users = [];
  try {
    for (let id of ids) {
      let user = await axios.get('...url...');
      users.push(user);
    }
  } catch (err) {...}
  return users;
}

const vips = await getUsers(...);
```

Pitfall #4: Reduced Parallelism

```
const axios = require('axios');
```

```
async function getUsers(ids) {
```

```
  let users = [];
```

```
  try {
```

```
    for (let id of ids) {
```

```
      let user = await axios.get('...url...');
```

```
      users.push(user);
```

```
    }
```

```
  } catch (err) {...}
```

```
  return users;
```

```
}
```

```
const vips = await getUsers(...);
```

- If order doesn't matter, why get `user` sequentially?

Parallel Awaiting

```
// get a user object; blocked  
let fu = await getFirstUser();
```

```
// get a promise immediately; async jobs starts  
let fp = getFirstUser();
```

```
// sequential awaiting  
let fu = await getFirstUser();  
let lu = await getLastUser();
```

```
// parallel awaiting  
let fp = getFirstUser(); // async jobs starts  
let lp = getLastUser(); // async jobs starts  
let [fu, lu] = await Promise.all([fp, lp]);
```

- `Promise.all()` creates a promise that resolves only when all child promises resolve

Solution

```
const axios = require('axios');

async function getUsers(ids) {
  let promises = [];
  try {
    for (let id of ids) {
      let promise = axios.get('...url...');
      promises.push(promise);
    }
  } catch (err) {...}
  return await Promise.all(promises);;
}
const vips = await getUsers(...);
```