CS 4530
Software Engineering

Lecture 7 - Asynchronous Programming II

Jonathan Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

Zoom Mechanics

- Recording: This meeting is being recorded
- If you feel comfortable having your camera on, please do so! If not: a photo?
- I can see the zoom chat while lecturing, slack while you're in breakout rooms
- If you have a question or comment, please either:
  - "Raise hand" - I will call on you
  - Write "Q: <my question>" in chat - I will answer your question, and might mention your name and ask you a follow-up to make sure your question is addressed
  - Write "SQ: <my question>" in chat - I will answer your question, and not mention your name or expect you to respond verbally

Today’s Agenda

Administrative:

- Team formation due Friday
- HW2 posted, due next Friday
- HW1 solution posted on Piazza

Today’s session:

- Review: Asynchronous Programming
- Activity: Asynchronous Programming

Review: Asynchronous Programming in JS/TS

Promises

`axios.get` returns a `Promise` for an `AxiosResponse`

```
console.log('Making a request to rest-example');
axios.get('https://rest-example.covey.town/') // axios is a popular library for making HTTP requests
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
console.log('Response sent!');
```

`Promise.then` will run the event handler provided once the value that is promised becomes available

Output:

```
Making a request to rest-example
Response sent!
Heard back from server
This is GET number 4 on the current server
```

`axios.get` is an **asynchronous call**

Review: Making lots of requests

3 Requests: What is the output?

```
console.log('Making a requests');
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
  });
console.log('Requests sent!');
```

These 2 lines ALWAYS first (same handler)

Sample Output:

```
Making a requests
Requests sent!
Heard back from Google
Heard back from server
This is GET number 6 on the current server
Heard back from Facebook
```

These 2 lines ALWAYS together (same handler)

No guarantee on order of hearing back from Google, our server, or Facebook (new handlers)

Review: Implications of run-to-completion

Run-to-completion: first 2 lines ALWAYS first, covey.town handler lines always together

```
console.log('Making a requests');
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
  });
console.log('Requests sent!');
```

These 2 lines ALWAYS first (same handler)

Sample Output:

```
Making a requests
Requests sent!
Heard back from Google
Heard back from server
This is GET number 6 on the current server
Heard back from Facebook
```

These 2 lines ALWAYS together (same handler)

No guarantee on order of hearing back from Google, our server, or Facebook (new handlers)

Review: What NOT to do in an event handler?

Run-to-completion: Slow handlers are really bad.

```
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
    fs.writeFileSync("google-response.txt",response.data);
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
    fs.writeFileSync("facebook-response.txt",response.data);
  });
```

3 seconds

Write a file *synchronously*
(write it in this event handler)

```
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
  console.log('Heard back from server');
  console.log(response.data);
});
axios.get('https://www.google.com/')
  .then((response) =>{
    console.log('Heard back from Google');
    return fsPromises.writeFile("google-response.txt",response.data);
  });
axios.get('https://www.facebook.com/')
  .then((response) =>{
    console.log('Heard back from Facebook');
    return fsPromises.writeFile("facebook-response.txt",response.data);
  });
```

2.1 seconds

Write a file a*synchronously*
(Ask NodeJS to write it in the background, this returns a new Promise to tell us when it's done)

Good news: You usually have to go out of your way to use synchronous I/O in NodeJS (the methods all have the word "Sync" in them)

Review: Async/Await

Your asynchronous friend

- Rules of the road:
  - You can only call **await** from a function that is **async**
  - You can only **await** on functions that return a **Promise**
  - Beware: **await** makes your code synchronous (this is what we want it for)!
  - Handle errors using try/catch

```
axios.get('https://rest-example.covey.town/').then(response => {
  console.log('Heard back from server');
  console.log(response.data);
}).catch(err => {
  console.log("Uh oh!");
  console.trace(err);
});
```

```
async function axiosAwaitExample() {
  try{
    const response = await axios.get('https://rest-example.covey.town/')
    console.log('Heard back from server');
    console.log(response.data);
  } catch(err){
    console.log("Uh oh!");
    console.trace(err);
  }
}
```

Review: Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics (`async/await` vs `Promise`)

```
function runClientPromises() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  const promisesForTranscripts = studentIDs.map(
    studentID => axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
      .then((response) =>
        fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
      ));
  return Promise.all(promisesForTranscripts).then(results => {
    const statsPromises = studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`));
    return Promise.all(statsPromises).then(stats => {
      const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
      console.log(`Finished calculating size: ${totalSize}`);
    });
  }).then(() => {
    console.log('Done');
  });
  console.log('Requests sent!');
}
```

```
async function runClientAsync() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  const promisesForTranscripts = studentIDs.map(
    async (studentID) => {
      const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
      await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
    });
  console.log('Requests sent!');
  await Promise.all(promisesForTranscripts);
  const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`)));
  const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
  console.log(`Finished calculating size: ${totalSize}`);
  console.log('Done');
}
```

New example: A bad handler

For large values of count, this is very slow!

```
function approximatePi(count) {
  let inside = 0;
  const r = 5;
  console.log(`Approximating Pi using ${count} iterations`)
  for (let i = 0; i < count; i++) {
    const x = Math.random() * r * 2 - r;
    const y = Math.random() * r * 2 - r;
    if ((x * x + y * y) < r * r) {
      inside++
    }
  }
  const ret = 4.0 * inside / count;
  console.log(`Computed: ${ret}`);
  return ret;
}
```

Review: Async/Await gone mad

Where you place awaits can make a big difference!

The code we've seen on past slides:

For each student: make an async handler to fetch their transcript and save it

```
on runClientAsync() {
g('Making a requests');
entIDs = [1, 2, 3, 4];
const promisesForTranscripts = studentIDs.map(
  async (studentID) => {
    const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
    await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
  });
console.log('Requests sent!');
await Promise.all(promisesForTranscripts);
const stats = await Promise.all(studentIDs.map(studentID => fsPromises.stat(`transcript-${studentID}.json`)));
const totalSize = stats.reduce((runningTotal, val) => runningTotal + val.size, 0);
console.log(`Finished calculating size: ${totalSize}`);
```

Running time: 1.5 sec

This does something different:

For each student: wait to fetch their transcript, then wait to write it, then go on to the next student

```
async function runClientAsyncSerially() {
  console.log('Making a requests');
  const studentIDs = [1, 2, 3, 4];
  for(let studentID of studentIDs){
    const response = await axios.get(`https://rest-example.covey.town/transcripts/${studentID}`);
    await fsPromises.writeFile(`transcript-${response.data.student.studentID}.json`, JSON.stringify(response.data))
  }
  let totalSize = 0;
  for(let studentID of studentIDs){
    const stats = await fsPromises.stat(`transcript-${studentID}.json`);
    totalSize += stats.size;
  }
  console.log(`Finished calculating size: ${totalSize}`);
}
```

Running time: 2.2 sec

This is what we mean by "your code can become synchronous"

Review: Async/Await Programming Activity

Transcript Server: Create a student, then update their

1. Create a new student in the transcript server

```
axios.post('https://rest-example.covey.town/transcripts', {name: 'Breakout Group 0'})
```

then…

2. Assign several grades for that student

```
axios.post(`https://rest-example.covey.town/transcripts/${studentID}/${course}`,{grade: theGrade}))
```

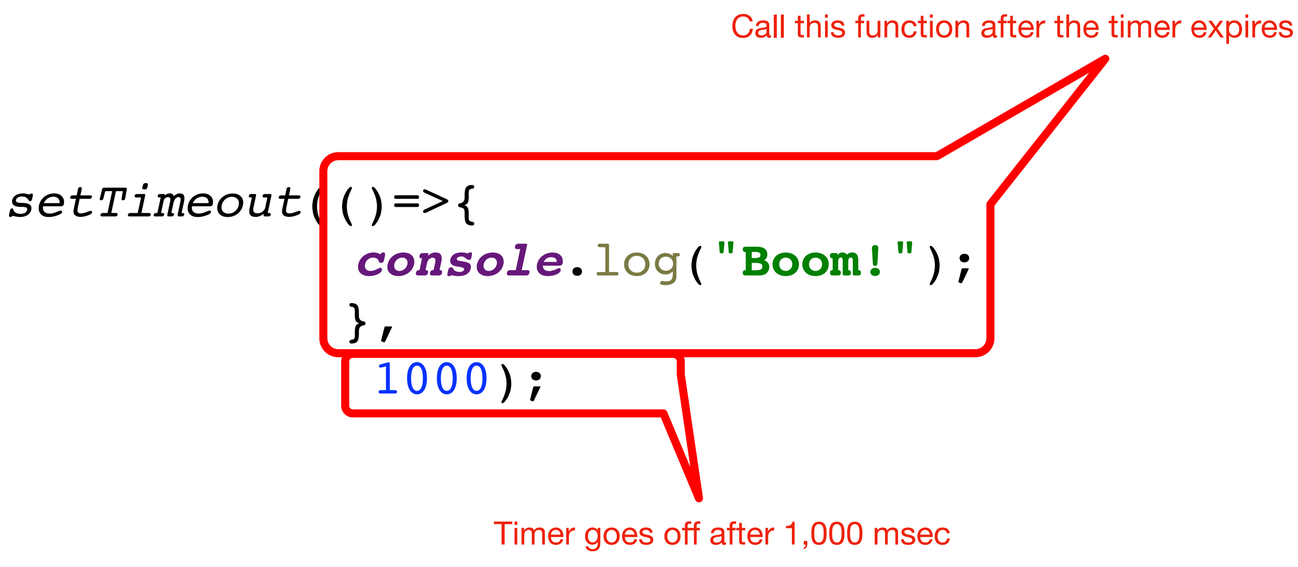
then…

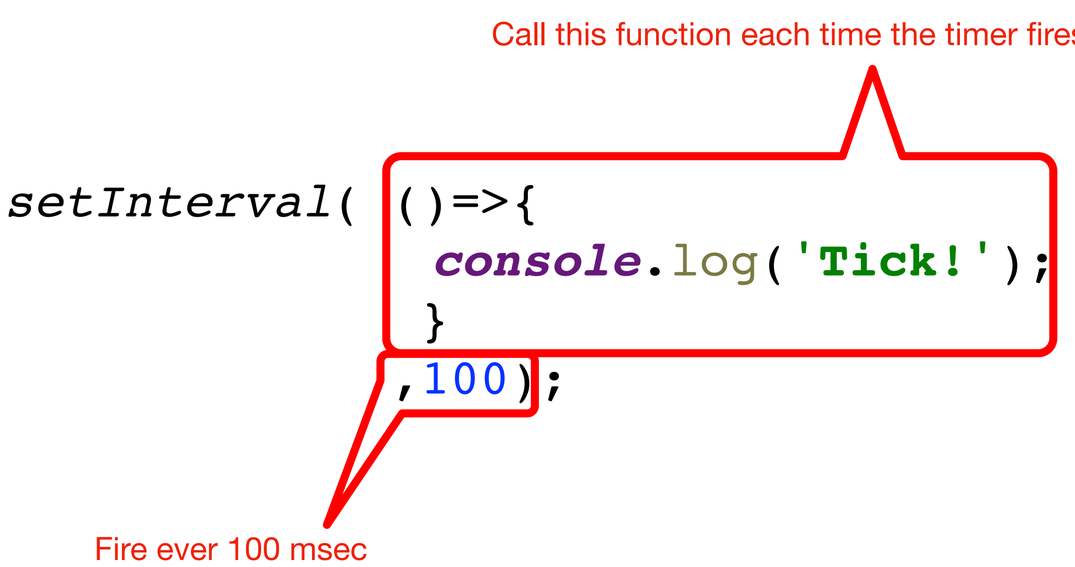
3. Fetch the transcript for that student

```
axios.get(`https://rest-example.covey.town/transcripts/${studentID}`)
```

If you finish with time to spare, try to make different variants: make a lot of requests concurrently vs making the requests synchronously (waiting between each request)

Scheduling Asynchronous Tasks: Timers






```
const timedBoom = setTimeout(()=>{
  console.log("Boom!");
}, 1000);
clearInterval(timedBoom) // Defuse Bomb
```

```
const ticker = setInterval(()=>{
  console.log('Tick!');
},100);
clearInterval(ticker) // Cancel timer
```

Writing our own Promises

Call this function to “resolve” the promise (whatever you pass to resolve gets passed to “then”)

```
function timedPromise(): Promise<number> {
  return new Promise<number>((resolve, reject) => {
    const random = Math.random();
    if (random < 0.5)
      setTimeout(() => {
        reject(random);
      }, 1000);
    else
      setTimeout(() => {
        resolve(random);
      }, 1000);
  });
}
timedPromise().then((val)=>{
  console.log(`Promise succeeded with ${val}`)
}).catch(val =>{
  console.error(`Promise failed with ${val}`)
})
```

Call this function to “reject” the promise (whatever you pass to reject gets passed to “catch”)

No matter how many times the “.then” is called, this code runs only once: when the Promise is created. Once resolve or reject is called, the value of the promise is locked-in

Asynchronous activity

Download this:

https://neu-se.github.io/CS4530-CS5500-Spring-2021/Examples/Example%204.0%20transcript-server-client.zip

Instructions in README.md

(zip is updated from Monday, if you downloaded previously, please re-download)

This work is licensed under a Creative Commons Attribution-ShareAlike license