

CS 4350: Fundamentals of Software Engineering

Lesson 4.2: Asynchronous Programming in TypeScript

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

Learning Goals for this Lesson

- At the end of this lesson, you should be able to:
 - Be able to write asynchronous code in TypeScript using both Promises and `async/await`
 - Understand how to achieve concurrency through asynchronous operations in TypeScript

Not all Asynchronous Code uses Await

Both code snippets are identical once JS engine compiles them

Using async/await

```
async function makeOneGetRequest(){
  const response = await axios.get('https://rest-example.covey.town');
  console.log('Heard back from server');
  console.log(response.data);
}
makeOneGetRequest();
console.log('Made Request');
```

We pass a function to “.then”, which is called after the promise is resolved

Directly using Promise

```
axios.get('https://rest-example.covey.town/')
  .then((response) =>{
    console.log('Heard back from server');
    console.log(response.data);
  });
console.log('Made Request');
```

Promises Enforce Ordering Through “Then”

Code after the async call runs *immediately*

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

Sample Output:

```
Making 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 first (same listener)

These 2 lines ALWAYS together (same listener)

No guarantee on order of hearing back from Google, our server, or Facebook

Each Listener Returns a Promise for Itself

Both examples produce the exact same output

```
async function makeOneGetRequest(){
  const response = await axios.get('https://rest-
example.covey.town');
  console.log(response.data);
}
```

```
console.log('Making first request');
makeOneGetRequest().then(() =>{
  console.log('Making second request');
  return makeOneGetRequest();
}).then(() => {
  console.log('Making third request');
  return makeOneGetRequest();
}).then(() =>{
  console.log('All done!');
});
```

```
async function makeThreeSerialRequests(){
1.  console.log('Making first request');
2.  await makeOneGetRequest();
3.  console.log('Making second request');
4.  await makeOneGetRequest();
5.  console.log('Making third request');
6.  await makeOneGetRequest();
7.  console.log('All done!');
}
makeThreeSerialRequests();
```

Syntax for Writing Asynchronous Code

For Async/Await and for Promises

- 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 instead of “catch” (common gotcha with promises)

```
async function makeOneGetRequest(): Promise<void> {  
  console.log("Making Request");  
  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);  
  }  
}
```

```
function makeOneGetRequestNoAsync(): Promise<void> {  
  console.log("Making Request");  
  return 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);  
  });  
}
```

Promise.all Allows for Concurrency

Promise.all creates one Promise for many

```
async function makeOneGetRequest(){
  const response = await axios.get('https://rest-example.covey.town');
  console.log(response.data);
}
```

```
async function makeThreeSerialRequests():
Promise<void> {
  await makeOneGetRequest();
  await makeOneGetRequest();
  await makeOneGetRequest();
  console.log('Heard back from all of the
requests');
}

makeThreeSerialRequests();
```

Output:

```
This is GET number 1 on the current server
This is GET number 2 on the current server
This is GET number 3 on the current server
Heard back from all of the requests
```

```
async function makeThreeGetRequests() {
  await Promise.all([
    makeOneGetRequest(),
    makeOneGetRequest(),
    makeOneGetRequest(),
  ]);
  console.log('Heard back from all of the requests');
}

makeThreeGetRequests();
```

Output:

```
This is GET number 3 on the current server
This is GET number 1 on the current server
This is GET number 2 on the current server
Heard back from all of the requests
```

Mask Latency With Concurrency

```
async function makeThreeSerialRequests():  
Promise<void> {  
  await makeOneGetRequest();  
  await makeOneGetRequest();  
  await makeOneGetRequest();  
  console.log('Heard back from all of the  
requests');  
}  
  
makeThreeSerialRequests();
```

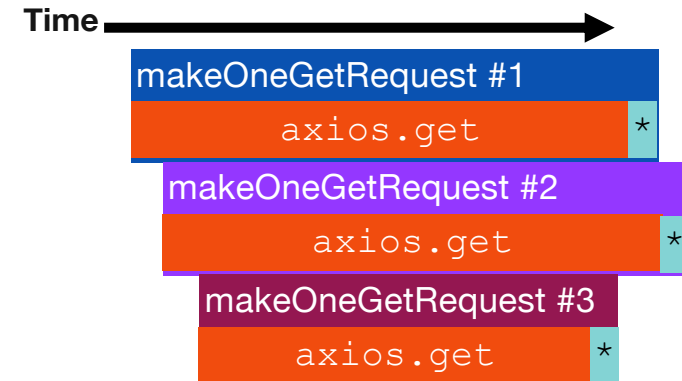
```
async function makeThreeGetRequests() {  
  await Promise.all([  
    makeOneGetRequest(),  
    makeOneGetRequest(),  
    makeOneGetRequest(),  
  ]);  
  console.log('Heard back from all of the requests');  
}  
  
makeThreeGetRequests();
```

Sequential version: ~200msec



*console.log

Concurrent version: ~70msec

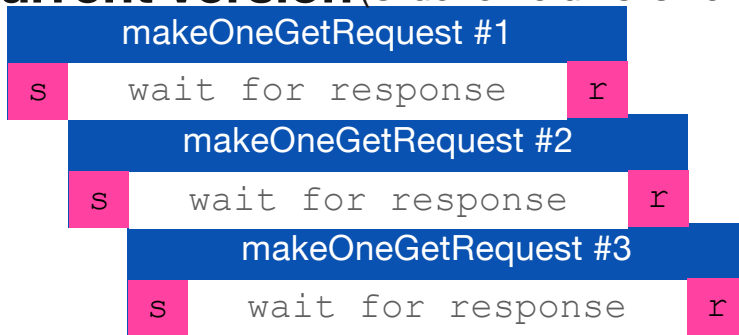


Mask Latency With Concurrency

Sequential version

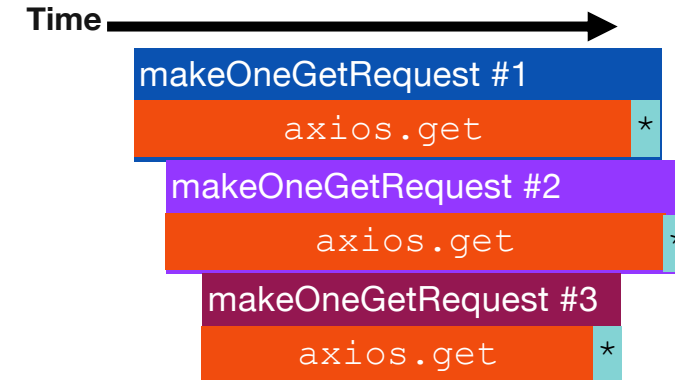
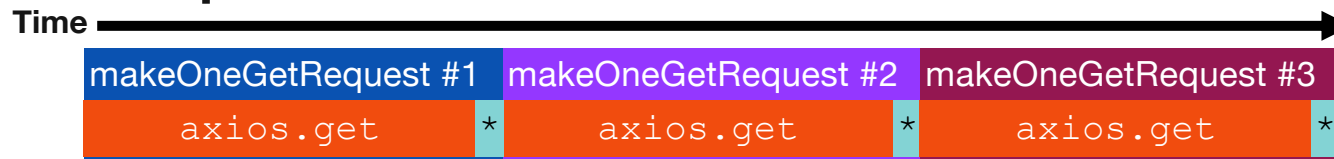


Concurrent version (Order of returns is non-deterministic)



Sequential version: ~200msec

Concurrent version: ~70msec



*console.log

Leverage Concurrency When Possible

Sequential version: ~200msec

```
async function makeThreeSerialRequests():  
Promise<void> {  
    await makeOneGetRequest();  
    await makeOneGetRequest();  
    await makeOneGetRequest();  
}  
  
makeThreeSerialRequests();
```

“Don’t make another request until you got the last response back”

Concurrent version: ~70msec

```
async function makeThreeGetRequests() {  
    await Promise.all([  
        makeOneGetRequest(),  
        makeOneGetRequest(),  
        makeOneGetRequest(),  
    ]);  
}  
  
makeThreeGetRequests();
```

“Make all of the requests at the same time, then wait for all of the responses”

Don't Perform Long-Running Computation in Asynchronous Code



```
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);
  });
```

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

3 seconds




```
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);
  });
```

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

2.1 seconds

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)



Don't Perform Long-Running Computation in Asynchronous Code

For large values of count, this will prevent anything from happening in JS until it's done!

```
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;  
}
```

General Rules for Writing Asynchronous Code

- Don't perform long-running computations or synchronous IO
- Leverage concurrency when possible
 - Remember that event events are processed in the order they are received
 - Events might arrive in unexpected order!
- Always check for errors (try/catch for async/await, “.catch” for promises)

A Full-Featured Asynchronous Example

“The Transcript Server” – A web service for us to play with

```
POST /transcripts
-- adds a new student to the database,
-- returns an ID for this student.
-- requires a body parameter 'name'
-- Multiple students may have the same name.
GET /transcripts/:ID
-- returns transcript for student with given ID. Fails if no such student
DELETE /transcripts/:ID
-- deletes transcript for student with the given ID, fails if no such student
POST /transcripts/:studentID/:courseNumber
-- adds an entry in this student's transcript with given name and course.
-- Requires a body parameter 'grade'
-- Fails if there is already an entry for this course in the student's transcript
GET /transcripts/:studentID/:courseNumber
-- returns the student's grade in the specified course.
-- Fails if student or course is missing.
GET /studentids?name=string
-- returns list of IDs for student with the given name
```

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
 - Request each student's transcript
 - Then for each transcript, save it to disk so that we have a copy
 - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
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}`)  
    });  
  console.log('Requests sent!');  
}
```

async: this function will automatically return a promise

Functional magic: map will apply the function specified to each element in the array and return a new array containing the result of each of those functions

await: wait for promise to resolve, then get its resolved value

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
 - Request each student's transcript
 - Then for each transcript, save it to disk so that we have a copy
 - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

async: the Promise we return won't be resolved until everything we await is

```
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: wait for promise to resolve,
then get its resolved value

Example: Writing Asynchronous Tasks

Transcript Server: Calculating statistics

- From an array of StudentIDs:
 - Request each student's transcript
 - Then for each transcript, save it to disk so that we have a copy
 - Then once all of the pages are downloaded and saved, print out the total size of all of the files that were saved

```
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');  
}
```

await for all transcripts to be downloaded and saved

await for all file statistics to be collected

Leverage Concurrency When Possible

Where you place awaits can make a big difference!

The code we've seen on past slides:

```
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}');  
}
```

Running time:
1.5 sec

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

This accomplishes the same function, but without concurrency:

```
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”

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

Async/Await Programming Activity

Transcript Server: Create a student, then post their grades

1. Create a new student in the transcript server

```
await client.addStudent('test student');  
then...
```

2. Assign several grades for that student

```
await client.addGrade(studentID, 'demo course', 100);  
then...
```

3. Fetch the transcript for that student

```
await client.getTranscript(studentID)
```

Your task will be to take a list of students with a list of grades, post them, and return all of the resulting transcripts

Download the activity (includes instructions in README.md):
Linked from course webpage for week 4, or at <https://bit.ly/34GbcN6>

Learning Goals for this Lesson

- At the end of this lesson, you should be able to:
 - Be able to write asynchronous code in TypeScript using both Promises and `async/await`
 - Understand how to achieve concurrency through asynchronous operations in TypeScript