# CS 4350: Fundamentals of Software Engineering
# CS 5500: Foundations of Software Engineering

## Lesson 4.3 Building a web client with async

Jon Bell, John Boyland, Mitch Wand

Khoury College of Computer Sciences

# Learning Objectives for this Lesson

- By the end of this lesson you should be able to:
  - Explain how a web client can be built in a layered fashion
  - Write scripts for the client that send out multiple http requests asynchronously.

# Outline of this Lesson

- Review:  promises and asynchronous functions
- Design of a scriptable web client using async

examples-4.2/example6.ts

# Review: Example with async/await

```
async function driver2(promiseName: string, flag: boolean) {
    try {
        console.log(`starting driver2(${flag})`)
        const n = await makePromise1(promiseName, flag, 10)
        console.log(`promise ${promiseName} fulfilled and passed ${n} to its suc
cessor`);
        const m = n + 1
        console.log(`the second then block received ${m}`)
    } catch (n) { console.log(`promise ${promiseName} rejected and passed "${n}"
 to its successor`) }
}
```

# Example without async/await

examples-4.2/example6.ts

```
function driver(promiseName: string, flag: boolean) {
    console.log(`starting driver(${flag})`)
    makePromise1(promiseName,flag,10)
        .then(n => {console.log(`promise ${promiseName} fulfilled and passed ${n}
 to its successor`);
                    return n+1
                })
        .then(m => console.log(`the second then block received ${m}`))
        .catch(n => console.log(`promise ${promiseName} rejected and passed "${n}
" to its successor`))
}
```

# Things to know about **async/await**

- An **async** function always returns a promise.
- Because a promise is created, it is automatically thrown in the pool of handlers to be run when ready
- The **async** keyword tells the compiler to do the translation
- Therefore, **await** makes no sense except in the body of an **async** function.
- The try/catch is optional, but typical.

# That was a long story to reach a simple conclusion

- A useful but complex pattern of behaviors is encapsulated in a single language construct.
- In the olden days, this might have been a "design pattern"
- Illustrates the power of programming-language technology

# Today's goal: A scriptable web client

- Usually you will interact with the web using a browser (e.g. using REACT)
- But you don't need that complexity immediately
- Today, we'll build a client that you can use to write scripts that interact with the transcript server.

# Goal: be able to write things like

```
async function script1() {
  try {
    console.log('starting script1()');
    const p1 = await ds.getTranscript(2);
    console.log('script1 says: getTranscript(2) says:', p1);
    const blakeIDs = await ds.getStudentIDs('blake');
    console.log('script1 says: students named blake:', blakeIDs);
    try {
      await (ds.addGrade(blakeIDs[0], 'cs101', 85));
    } catch {console.log("script1 says: blake's grade already there, continuing");}
    console.log('script1 says', await ds.getGrade(blakeIDs[0], 'cs101'));
    console.log('script1 succeeded');
  } catch { console.log('script1 failed'); }
}
```

Here we've written a whole script of interactions with the server. We use 'await' to make sure that everything is done in the exact order we want.

We've made this script especially verbose so you can see the details of what's going on here. Probably good for debugging, maybe not so much when deployed.

# And things like this

```
async function getTranscriptsByName(studentName: string) {
  try {
    console.log(`starting getTranscriptsByName(${studentName})`);
    const ids = await ds.getStudentIDs(studentName);
    // put out all the requests in parallel, not sequentially
    // 'requests' becomes bound to an array of promises
    const requests : Promise<Transcript>[] = ids.map(id => ds.getTranscript(id));
    const transcripts = await Promise.all(requests);
    console.log(`getTranscriptsByName says: ${studentName}'s transcripts:`,
                transcripts);
    console.log('getTranscriptsByName succeeded');
  } catch {
    console.log('getTranscriptsByName failed');
  }
}

getTranscriptsByName('blake')
```

# Our client uses a layered architecture

index.ts : contains scripts to be executed.
Calls: getTranscript, getStudentIDs, etc., corresponding to the REST endpoints

dataService.ts: provides REST endpoints
exports: getTranscript, getStudentIDs, etc.

remoteService.ts : provides http methods
exports: remoteGet, remotePost, etc.

This is the only module that refers to axios. So if we switch to another http package, this is the only file that needs changing

axios: an npm package that actually does the http work
provides: axios.get, axios.post, etc

# remoteService.ts

3 functions, one per http Method.

```
import axios, { AxiosResponse } from 'axios';

axios.defaults.baseURL = 'http://localhost:4001'; // where to send the requests

export async function remoteGet<T>(path:string) : Promise<T> {
  try {
    const response : AxiosResponse<T> = await axios.get(path);
    return (response.data);
  } catch (e) { throw new Error(e); }
}

export async function remoteDelete<T>(path:string) : Promise<T> {
  // similar
}

export async function remotePost<T>(path:string, data?:T) : Promise<T> {
  try {
    const response : AxiosResponse<T> = await axios.post(path, data);
    return (response.data);
  } catch (e) { throw new Error(e); }
}
```

# dataService.ts

One function per REST endpoint. (Similar to transcriptManager.ts)

```
import { StudentID, Course, Transcript } from './types';
import { remoteGet, remoteDelete, remotePost } from './remoteService';

// POST /transcripts
export async function addStudent(studentName: string): Promise<{ name: string }> {
  return remotePost('/transcripts', { name: studentName });
}

// GET  /studentids?name=string     -- returns list of IDs for student with the given name
export async function getStudentIDs(studentName:string) : Promise<StudentID[]> {
  return remoteGet(`studentids?name=${studentName}`);
}

/* POST /transcripts/:studentID/:course   -- /transcripts
Requires a post parameter 'grade'.
Fails if there is already an entry for this course in the student's transcript
*/
export async function addGrade(studentID: StudentID, course: Course, grade: number)
  : Promise<{ grade: number }> {
  return remotePost(`/transcripts/${studentID}/${course}`, { grade: grade });
}

// ...
```

# Running the system

- Unpack and install transcript-server and transcript-client
- In one shell, open the directory containing the server and run 'npm start'.  This will start the server and display the server console.
- In another shell, open the directory containing the client.  To run the client, say 'npm start'.
- You can always change the scripts in the client's index.ts

# Watching the system run

- Server startup messages:

```
> tsc && node ./dist/index.js

Initial list of transcripts:
[
  { student: { studentID: 1, studentName: 'avery' }, grades: [] },
  { student: { studentID: 2, studentName: 'blake' }, grades: [] },
  { student: { studentID: 3, studentName: 'blake' }, grades: [] },
  { student: { studentID: 4, studentName: 'casey' }, grades: [] }
]
Express server now listening on localhost:4001
```

- You are now watching the server log.
- Our server prints lots of log messages so you can see what's happening on the server side.

# index.ts (the actual client)

```
import * as ds from './dataService';
import { Transcript } from './types';

console.log('starting index.ts');
async function script1() {...}
async function getTranscriptsByName(studentName: string) {...}

// this creates a new promise that will executed when the scheduler
// gets around to it
script1();
// getTranscriptsByName() creates a promise that will be fulfilled later.
// But the console.log runs right away. So it prints "Promise<pending>"
console.log(getTranscriptsByName('blake'));

// also done right away
console.log('index.ts done');
```

# Client's output

```
starting index.ts
starting script1()
starting getTranscriptsByName(blake)
Promise { <pending> }
index.ts done
script1 says: getTranscript(2) says: { student: { studentID: 2,
studentName: 'blake' }, grades: [] }
getTranscriptsByName says: blake's transcripts: [
  { student: { studentID: 2, studentName: 'blake' }, grades: []
},
  { student: { studentID: 3, studentName: 'blake' }, grades: []
}
]
getTranscriptsByName succeeded
script1 says: students named blake: [ 2, 3 ]
script1 says { studentID: 2, course: 'cs101', grade: 85 }
script1 succeeded
```

Notice that script1 and getTranscriptbyName are running in separate handlers, so the promises they create are interleaved.

The promises generated by getTranscriptsByName are also asynchronous and may be interleaved, but that's a little harder to illustrate.

# Review: Learning Objectives for this Lesson

- You should now be able to:
  - Explain how a web client can be built in a layered fashion
  - Write scripts for the client that send out multiple http requests asynchronously.

# Next Steps

- Be prepared to explain each line in the output examples
- Create some new scripts like the ones here and try to predict what they will do.
- Add some new REST endpoints to the server and the client.
- Think of some good questions to bring to class!