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

Lesson 3.4: Building a REST server

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 prepared to:
  - Explain the structure of a server in express.js
  - Define 'middleware' and 'route' in the context of an express.js server
  - Build a server for a simple REST protocol in express.js

Outline of this Lesson

1. Review of REST
2. Discussion of the protocol for our example
3. Demo of the server
4. Structure of a server in express
5. Codewalk of the server, with discussion of the development process.

1. REST Principles

- Single Server
  - Client calls server, server responds. That's it.
  - Separation of concerns: client doesn't worry about data, server doesn't worry about UI
  - Server may pass request on to other machines, but that's not visible to the client
- Stateless
  - No session state in the server
  - Each client request must contain all the information the server needs to process the request
- Uniform Interface
  - associate URIs with resources
- Uniform Cacheability
  - requests must classify themselves as cacheable or not.

Uniform Interface: Nouns are represented as URIs

- In a RESTful system, the server is visualized as a store of resources (nouns), each of which has some data associated with it.
- URIs represent these resources
- Examples:
  - `/cities/losangeles`
  - `/transcripts/00345/graduate` (student 00345 has several transcripts in the system; this is the graduate one)
- Anti-examples:
  - `/getCity/losangeles`
  - `/getCitybyID/50654`
  - `/Cities.php?id=50654`

We prefer plural nouns for toplevel resources, as you see here.

Useful heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like? But you don't have to actually keep the data in that way.

Verbs are represented as http methods

- In REST, there are exactly four things you can do with a resource:
- POST: request the server to create a resource
    - there are several ways in which the value for the new resource can be transmitted (more In a minute)
- GET: request the server to respond with a representation of the resource
- PUT: request the server to replace the value of the resource by the given value
- DELETE: request the server to delete the resource

Associating parameters with a request

There are at least 3 ways to associate parameters with a request:

- path parameters. These specify portions of the path to the resource. For example, your REST protocol might allow a path like

  ```
  /transcripts/00345/graduate
  ```

- query parameters. These are part of the URI and are typically used as search items. For example, your REST protocol might allow a path like

  ```
  /transcripts/graduate?lastname=covey&firstname=avery
  ```

- body parameters. These occur in the body of the request. They could be formatted in JSON or www-urlencoded (like our query parameters above) or anything else. This choice is up to the protocol designer.

The Protocol for our example

Remember the heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like?

```
POST /transcripts
 -- adds a new student to the database,
 -- returns an ID for this student.
 -- requires a body parameter 'name', url-encoded (eg name=avery)
 -- 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', url-encoded
 -- 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
```

Reflects a different view of the data

Development: first we built our information store

- Just because it looks like a hierarchical structure doesn't mean you must represent it that way.
- We wrote a TypeScript module (file) called transcriptManager.ts that exported some types:

```
export type StudentID = number
export type Student = {studentID: number, studentName: string}
export type Course = string
export type CourseGrade = {course:Course, grade:number}
export type Transcript = {student:Student, grades:CourseGrade[]}
```

- and a bunch of functions:

CourseName probably would have been a better name than Course, sorry. What other names could have been improved here?

Functions exported by transcriptManager (1)

```
// initializes the database with 4 students,
// each with an empty transcript (handy for debugging)
export function initialize ()

// returns a list of all the transcripts.
// handy for debugging
export function getAll()

// creates an empty transcript for a student with this name,
// and returns a fresh ID number
export function addStudent(name:string) : StudentID

// gets transcript for given ID.  Returns undefined if missing
export function getTranscript(studentID:number) : Transcript

// returns list of studentIDs matching a given name
export function getStudentIDs(studentName:string) : StudentID[]
```

Pretty much one function to implement each of the actions in the protocol.

Functions exported by transcriptManager (2)

```
// deletes student with the given ID from the database.
// throws exception if no such student.
export function deleteStudent(studentID:StudentID)

// adds a grade for the given student in the given course.
// throws error if student already has a grade in that course.
export function addGrade(studentID: StudentID, course: Course, grade: number)

// returns the grade for the given student in the given course
// throws an error if no such student or no such grade
export function getGrade(studentID:StudentID, course:Course) : number
```

Could I have made this into a singleton with an interface that looks like this? Sure, but there doesn't seem to be any advantage to doing so.

Testing the transcriptManager

- I also wrote some tests for the transcript manager.
- I'm a pretty good coder, so I didn't have too many tests initially.
- However, when things went wrong, the first thing I did was to add some tests to transcriptManager to make sure that it was sending the right data back to the server.

Starting the server

```
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ npm install
<bunches of stuff...>

wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ npm run run

> transcript-server@2.0.0 run C:\Users\wand\Demos\Transcript-Server
> 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
```

Interacting with the server from the command line

We'll use 'curl' to send requests to the server

```
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ curl -s -X GET localhost:4001/transcripts/3
{"student":{"studentID":3,"studentName":"blake"},"grades":[]}
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ curl -s -X POST localhost:4001/transcripts/3/cs100 -d grade=85
OK
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ curl -s -X GET localhost:4001/transcripts/3/cs100
{"studentID":3,"course":"cs100","grade":85}
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ curl -s -X GET localhost:4001/transcripts/3
{"student":{"studentID":3,"studentName":"blake"},"grades":[{"course":"cs100
","grade":85}]}
```

Student #3 was blake. Get his transcript

POST an 85 in cs100 for blake. The –d tells curl to put this info in the

What was blake's grade in cs100? Answer comes back in JSON

Look at blake's whole transcript

Interacting with the server from the command line

```
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ curl -s -X POST localhost:4001/transcripts -d name=zeta
{"studentID":5}
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ curl -s -X GET localhost:4001/transcripts/5
{"student":{"studentID":5,"studentName":"zeta"},"grades":[]}
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$ curl -s -X GET localhost:4001/studentids?name=blake
[2,3]
wand@lenovo-2017 MINGW64 ~/Demos/Transcript-Server
$
```

Create a new student called zeta Server responds with their id, wrapped in JSON.

Look at zeta's transcript

Which students are named 'blake'?

Structure of an express server

Here is the null server. It listens on a port, but does nothing else.

```
import * as express from 'express'

// create the server, call it app
const app: express.Application = express();

// the port to listen on
const inputPort = 4001

// initialize the server
function initializeServer () {
    console.log(`Express server now listening on localhost:${inputPort}`)
}

// start the server listening on 4001
app.listen(inputPort,initializeServer)

// middleware
// routes
```

Create an instance of the server

and a create SPOC for the input port

Remember? Single Point of Control!

You could do other things here, like initialize the database.

Tell the server to run initializeServer() and start listening on the input port

middleware and routes will go here

Install the middleware

```
// middleware

// allow requests from any port or source.
import * as cors from 'cors'
app.use(cors())

// for parsing application/json
app.use(express.json());

// for parsing application/x-www-form-urlencoded
// converts foo=bar&baz=quux to {foo: 'bar', baz: 'quux'}
app.use(express.urlencoded({ extended: true }));
```

express creates a pipeline of plugins to parse and otherwise process the requests as they come in. These plugins are called 'middleware'.
app.use() installs a plugin in the processing pipeline

The headers in each request determine which parsers are invoked .

You will see blog posts telling you to install 'body-parser'. Don't bother; this is now part of express itself.

Install a route for each request in the protocol

```
// GET /transcripts
app.get('/transcripts', (req,res) => {
  console.log('Handling GET/transcripts')
  let data = db.getAll()
  console.log(data)
  res.status(200).send(data)
})
```

The routes follow the middleware. Each route is a handler for requests whose path matches the path in the route.

Here we've installed a handler for requests of the form GET /transcripts.

In the response, set the status to 200 ("OK") and send the data.

The second argument to app.get is always a function of two arguments: the request and the response. Here we don't need to look any closer at the request

I put in console.log(data) so I could check what was going on by looking at the server console. You might or might not want to keep this level of logging when you hand in a server.

POST /transcripts

Here we've installed a handler for requests of the form POST /transcripts.

```
app.post('/transcripts', (req,res) => {
  // use req.body.name to get the value of the post parameter
  // (in the body)
  const studentName : string = req.body.name;


  let studentID = db.addStudent(studentName)
  console.log(`Handling POST/transcripts name=${ )
  res.status(200).send({studentID: studentID})
})
```

Use req.body.<whatever> to get the value of a post parameter in the body

Call the database to add the student to the database.

Send some JSON back to the client. The argument to send must be either a string or some JSON; not a number.

What if the name parameter is missing? Good question!

ALWAYS need to worry about that! Our later routes are more careful...

Finger exercise: fix this route to do something more reasonable.

GET /transcripts/:id

```
// GET  /transcripts/:id           --
// returns transcript for student with given ID.
// Fails with a 404 if no such student
// req.params will look like {"id": 301}

app.get('/transcripts/:id', (req,res) => {
  // req.params to get components of the path (eg: /transcripts/301)
  const id = req.params.id
  console.log(`Handling GET /transcripts/:id id = ${id}`)
  const theTranscript = db.getTranscript(parseInt(id));
  if (theTranscript === undefined) {
    res.status(404).send(`No student with id = ${id}`)
  } {
    res.status(200).send(theTranscript)
  }
})
```

Use req.params to extract the 301 from /transcripts/301 .

Here we are relying on the error behavior of getTranscript.

What do you think will happen if we do a GET /transcripts/xyzzy ? Why? Try the experiment and see what happens. Can you explain why the server behaves the way it does?

GET /studentids?name=theName

```
app.get('/studentids', (req, res) => {
  // use req.query to get value of the parameter
  const studentName = req.query.name as string
  console.log(`Handling GET /studentids?name=${studentName}`)
  const ids = db.getStudentIDs(studentName)
  console.log(`ids = ${ids}`)
  res.status(200).send(ids)
})
```

use req.query to get the values of query parameters

If the original request is
/studentids?name=blake
then the value of req.query will be
{name: "blake"}

DELETE /transcripts/:id

```
// DELETE /transcripts/:ID
// deletes transcript for student with the given ID,
// fails with 404 if no such student
app.delete('/transcripts/:id', (req,res) => {
  const id = parseInt(req.params.id)
  console.log(`Handling DEL /transcripts, id = ${id}`)
  try {
    db.deleteStudent(id);
    res.sendStatus(200)
  } catch (e) {
    res.status(404).send(e.toString())
  }
})
```

You should surround anything that might fail in a try/catch.
You can transmit your own error message or rely on e.toString() .

Default routes

```
app.get('/:request*', (req, res) => {
  console.log(defaultErrorMessage('GET',req.params.request))
  res.sendStatus(404)
})

app.post('/:request*', (req, res) => {
  console.log(defaultErrorMessage('POST',req.params.request))
  res.sendStatus(404)
})

// etc.

function defaultErrorMessage
  (method: string, request: string): string
  {
    return `unknown ${method} request "${request}"`
  }
```

Any request that does not match any of the routes will reach Express's default 404 catcher.
I found it useful to have my own default routes, so I would be sure what was going on.

Review: Outline of this Lesson

1. Review of REST
2. Discussion of the protocol for our example
3. Demo of the server
4. Structure of a server in express
5. Codewalk of the server, with discussion of the development process.

Review: Learning Objective for this Lesson

- You should now be prepared to:
  - Explain the structure of a server in express.js
  - Define 'middleware' and 'route' in the context of an express.js server
  - Build a server for a simple REST protocol in express.js

Next steps...

- Come to class, prepared with questions!
- Think of some ways in which this transcript server might be improved or extended.