SERVER-SIDE

NODE JS II

EXPRESS JS

SEQUELIZE ORM

ADVANCED JAVASCRIPT

VERSION

v1.2 13 November 2021

v1.1 9 November 2021

v1.0 5 November 2021

ACKNOWLEDGMENTS

THANKS TO:

- Hamzeh Roumani, who has shaped EECS-4413 into a leading hands-on CS course at EECS and who generously shared all of his course materials and, more importantly, his teaching philosophy with me;
- Parke Godfrey, my long-suffering Master's supervisor and mentor; and
- Suprakash Datta for giving me this opportunity to teach this course.

PRINTABLE VERSION OF THE TALK

Download PDF

EXPRESSJS

ABOUT EXPRESS

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is an open source framework developed and maintained by the Node.js foundation.

ExpressJS is a web application framework that provides you with a simple API to build websites, web apps and back ends. With ExpressJS, you need not worry about low level protocols, processes, etc. Express provides a minimal interface to build our applications. It provides us the tools that are required to build our app. It is flexible as there are numerous modules available on npm, which can be directly plugged into Express. Express was developed by TJ Holowaychuk and is maintained by the Node.js foundation and numerous open source contributors.

EXPRESS ROUTING

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on). Each route can have one or more handler functions, which are executed when the route is matched. Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER)
```

Where:

- `app` is an instance of express.
- `METHOD` is an HTTP request method, in lowercase.
- `PATH` is a path on the server.
- `HANDLER` is the function executed when the route is matched.

```
1  const express = require('express');
2  const app = express();
3
4  // Respond with Hello World! on the homepage:
5  app.get('/', function (req, res) {
6    res.send('Hello World!');
7  });
8
9  // Respond to POST request on the root route (/), the app's home page:
10 app.post('/', function (req, res) {
11   res.send('Got a POST request');
12 });
13
14 // Respond to a PUT request to the /user route:
15 app.put('/user', function (req, res) {
16   res.send('Got a PUT request at /user');
17 });
18
19 // Respond to a DELETE request to the /user route:
20 app.delete('/user', function (req, res) {
21   res.send('Got a DELETE request at /user');
22 });
23
24 app.listen(3000);
```

EXPRESS STATIC FILES

To serve static files such as images, CSS files, and JavaScript files, use the `express.static` built-in middleware function in Express.

For example, use the following code to serve images, CSS files, and JavaScript files in a directory named `public`:

```
app.use(express.static('public'));
```

Now, you can load the files that are in the public directory:

```
http://localhost:3000/images/kitten.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/images/bg.png
http://localhost:3000/hello.html
```

To use multiple static assets directories, call the `express.static` middleware function multiple times:

```
app.use(express.static('public'));
app.use(express.static('files'));
```

To create a virtual path prefix (where the path does not actually exist in the file system) for files that are served by the `express.static` function, specify a mount path for the `static` directory, as shown below:

```
app.use('/static', express.static('public'));
```

Now, you can load the files that are in the `public` directory from the `/static` path prefix.

```
http://localhost:3000/static/images/kitten.jpg
http://localhost:3000/static/css/style.css
http://localhost:3000/static/js/app.js
http://localhost:3000/static/images/bg.png
http://localhost:3000/static/hello.html
```

However, the path that you provide to the express.static function is relative to the directory from where you launch your node process. If you run the express app from another directory, it's safer to use the absolute path of the directory that you want to serve:

```
const path = require('path');
app.use('/static', express.static(path.join(__dirname, 'public')));
```

EXPRESS SESSION

A website is based on the HTTP protocol. HTTP is a stateless protocol which means at the end of every request and response cycle, the client and the server forget about each other. This is where the session comes in. A session will contain some unique data about that client to allow the server to keep track of the user's state. In session-based authentication, the user's state is stored in the server's memory or a database.

```
$ npm install express-session
```

To initialize the session, we will set the session middleware inside the routes of the individual HTTP requests. When a client sends a request, the server will set a session ID and set the cookie equal to that session ID. The cookie is then stored in the set cookie HTTP header in the browser. Every time the browser (client) refreshes, the stored cookie will be a part of that request.

Resources:

- Session Management in Node.js using ExpressJS and Express Session
- express-session - npm
- ExpressJS - Sessions

```
1  const express = require('express');
2  const session = require('express-session');
3  const app = express();
4
5  // Use the session middleware
6  app.enable('trust proxy');
7  app.use(session({
8    secret: "secret",
9    resave: true,
10   saveUninitialized: true,
11   proxy: true
12 }));
13
14 app.get('/', (req, res) => {
15   if (req.session.page_views) {
16     req.session.page_views++;
17     res.send("You visited this page " + req.session.page_views + " times");
18   } else {
19     req.session.page_views = 1;
20     res.send("Welcome to this page for the first time!");
21   }
22 });
23
24 app.listen(0);
```

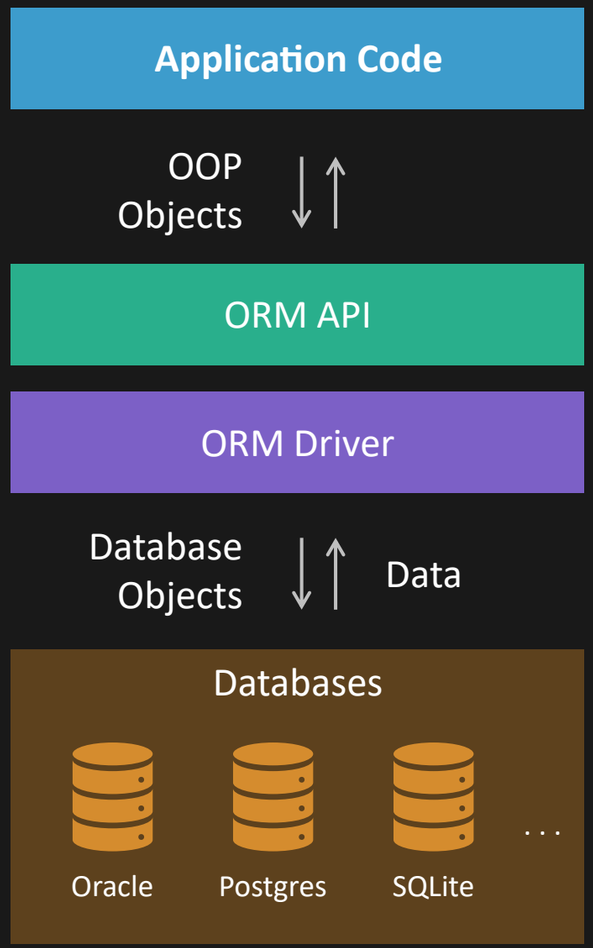
SEQUELIZE

OBJECT RELATIONAL MAPPING

OBJECT RELATIONAL MAPPING

Object-Relational Mapping (ORM) is a technique that lets you query and manipulate data from a database using an object-oriented paradigm. When talking about ORM, most people are referring to a library that implements the Object-Relational Mapping technique, hence the phrase “an ORM”.

An ORM library is a completely ordinary library written in your language of choice that encapsulates the code needed to manipulate the data, so you don’t use SQL anymore; you interact directly with an object in the same language you’re using.



KEY FEATURES

Some of the key features are as follows:

- It makes the application independent of the database management system being used in the backend, and so you can write a generic query. In case of migrating to another database, it becomes fairly a good deal to have ORM implemented in the project.
- Hassles of coders are reduced to learn SQL syntaxes separately for whichever database being used to support the application. Coders can shift their focus on optimizing the code and improving performance rather than dealing with connectivity issues.
- All small or big changes can be carried out via ORM, so there are no such restrictions when we deal with data. For example, JDBC comes with a lot of restrictions on extracting a result-set, process it and then commit it back to the database. This is not the case with ORMs. Even a single cell in the database can be retrieved, changed and saved back.
- The connection becomes robust, secure as there will be less intervention in code. It will handle all the necessary configurations required to map application programming language with the database's query language since there will be lesser intervention promoting secure application as a whole.
- There is a fairly large deal of ORMs present in the market as per the application language used. One can choose easily as per business requirements.
- There is an attached disadvantage in using ORM as well. That is when the database is in legacy file systems and disarranged. It becomes a task to arrange a whole lot of data and then map this with ORM. It is thereby suggested to use ORM when the back end is fairly managed.

PROS AND CONS

Using ORM saves a lot of time because:

- DRY: You write your data model in only one place, and it's easier to update, maintain, and reuse the code.
- A lot of stuff is done automatically, from database handling to I18N.
- It forces you to write MVC code, which, in the end, makes your code a little cleaner.
- You don't have to write poorly-formed SQL (most Web programmers really suck at it, because SQL is treated like a "sub" language, when in reality it's a very powerful and complex one).
- Sanitizing; using prepared statements or transactions are as easy as calling a method.

Using an ORM library is more flexible because:

- It fits in your natural way of coding (it's your language!).
- It abstracts the DB system, so you can change it whenever you want.
- The model is weakly bound to the rest of the application, so you can change it or use it anywhere else.
- It lets you use OOP goodness like data inheritance without a headache.

PROS AND CONS

Some of other highlights of ORM are:

- No need to learn a database query language.
- It propagates the idea of data abstraction, thus improving data security.
- Instead of storing big procedures in pl/SQL in the backend, these can be saved in the frontend. It improves flexibility to make changes.
- If there are many relations like 1: m, n: m and lesser 1:1 in database structure, then ORM works well.
- It reduces the hassles for coders by reducing the database query part handling.
- Queries via ORM can be written irrespective of whatever database one is using in the back end. This provides a lot of flexibility to the coder. This is one of the biggest advantages offered by ORMs.
- These are available for any object-oriented language, so it is not only specific to one language.

But ORM can be a pain:

- You have to learn it, and ORM libraries are not lightweight tools;
- You have to set it up. Same problem.
- Performance is OK for usual queries, but a SQL master will always do better with his own SQL for big projects.
- It abstracts the DB. While it’s OK if you know what’s happening behind the scene, it’s a trap for new programmers that can write very greedy statements, like a heavy hit in a for loop.

ADVANCED JAVASCRIPT

JAVASCRIPT CALLBACKS

In JavaScript, a callback is a function passed into another function as an argument to be executed later.

```
1  function download(url) {
2    setTimeout(() => {
3      console.log(`Downloading ${url} ...`);
4    }, 3 * 1000);
5  }
6
7  function process(picture) {
8    console.log(`Processing ${picture}`);
9  }
10
11 let url = 'picture.png';
12
13 download(url);
14 process(url);
```

WRONG ORDER

```
Processing picture.png
Downloading picture.png ...
```

Source: JavaScript Callbacks

JAVASCRIPT CALLBACKS

In JavaScript, a callback is a function passed into another function as an argument to be executed later.

```
1  function download(url, callback) {
2    setTimeout(() => {
3      console.log(`Downloading ${url} ...`);
4      callback(url);
5    }, 3000);
6  }
7
8  function process(picture) {
9    console.log(`Processing ${picture}`);
10 }
11
12 download('picture.png', process);
```

FIXED

```
Downloading picture.png ...
Processing picture.png
```

Source: JavaScript Callbacks

JAVASCRIPT CALLBACKS

In JavaScript, a callback is a function passed into another function as an argument to be executed later.

```
1  function download(url, callback) {
2    setTimeout(() => {
3      console.log(`Downloading ${url} ...`);
4      callback(url);
5    }, 3000);
6  }
7
8  download('picture.png', function (picture) {
9    console.log(`Processing ${picture}`);
10 });
```

FIXED

```
Downloading picture.png ...
Processing picture.png
```

Source: JavaScript Callbacks

JAVASCRIPT CALLBACKS

In JavaScript, a callback is a function passed into another function as an argument to be executed later.

```
1  function download(url, success, failure) {
2    setTimeout(() => {
3      console.log(`Downloading ${url} ...`);
4      // ... download ...
5      if (status === 200) {
6        success(url);
7      } else {
8        failure(url);
9      }
10   }, 3000);
11 }
12
13 download('picture.png',
14   (picture) => console.log(`Processing ${picture}`),
15   (picture) => console.log(`Handling error...`)
16 );
```

HANDLING ERRORS

```
Downloading picture.png ...
Processing picture.png

Downloading picture.png ...
Handling error...
```

Source: JavaScript Callbacks

JAVASCRIPT CALLBACKS

In JavaScript, a callback is a function passed into another function as an argument to be executed later.

```
1  function download(url, callback) {
2    setTimeout(() => {
3      console.log(`Downloading ${url} ...`);
4      callback(url);
5    }, 3000);
6  }
7
8  download('picture1.png', (picture) => {
9    console.log(`Processing ${picture}`);
10   download('picture2.png', (picture) => {
11     console.log(`Processing ${picture}`);
12     download('picture3.png', (picture) => {
13       console.log(`Processing ${picture}`);
14     });
15   });
16 });
```

NESTING CALLBACKS

```
Downloading picture1.png ..
Processing picture1.png
Downloading picture2.png ..
Processing picture2.png
Downloading picture3.png ..
Processing picture3.png
```

Source: JavaScript Callbacks

JAVASCRIPT CALLBACKS

In JavaScript, a callback is a function passed into another function as an argument to be executed later.

```
asyncFunction(function () {
  asyncFunction(function () {
    asyncFunction(function () {
      asyncFunction(function () {
        asyncFunction(function () {
          // ...
        });
      });
    });
  });
});
```

CALLBACK HELL

Nesting many asynchronous functions inside callbacks is known as the **Pyramid of Doom** or **Callback Hell**:

To avoid the pyramid of doom, you use promises or async/await functions.

Source: JavaScript Callbacks

JAVASCRIPT CLOSURE

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

```
function makeFunc() {
  let name = 'Mozilla'; // name is a local variable
  return function () { // the inner function, a closure
    alert(name); // use variable in parent function
  };
}

let myFunc = makeFunc();
myFunc();
```

```
function makeCounter() {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment() {
      changeBy(1);
    },
    decrement() {
      changeBy(-1);
    },
    value() {
      return privateCounter;
    }
  };
}

let counter1 = makeCounter();
let counter2 = makeCounter();

alert(counter1.value()); // 0

counter1.increment();
counter1.increment();
alert(counter1.value()); // 2

counter1.decrement();
alert(counter1.value()); // 1
alert(counter2.value()); // 0
```

PROMISE API

A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

`pending`
: Initial state, neither fulfilled nor rejected.

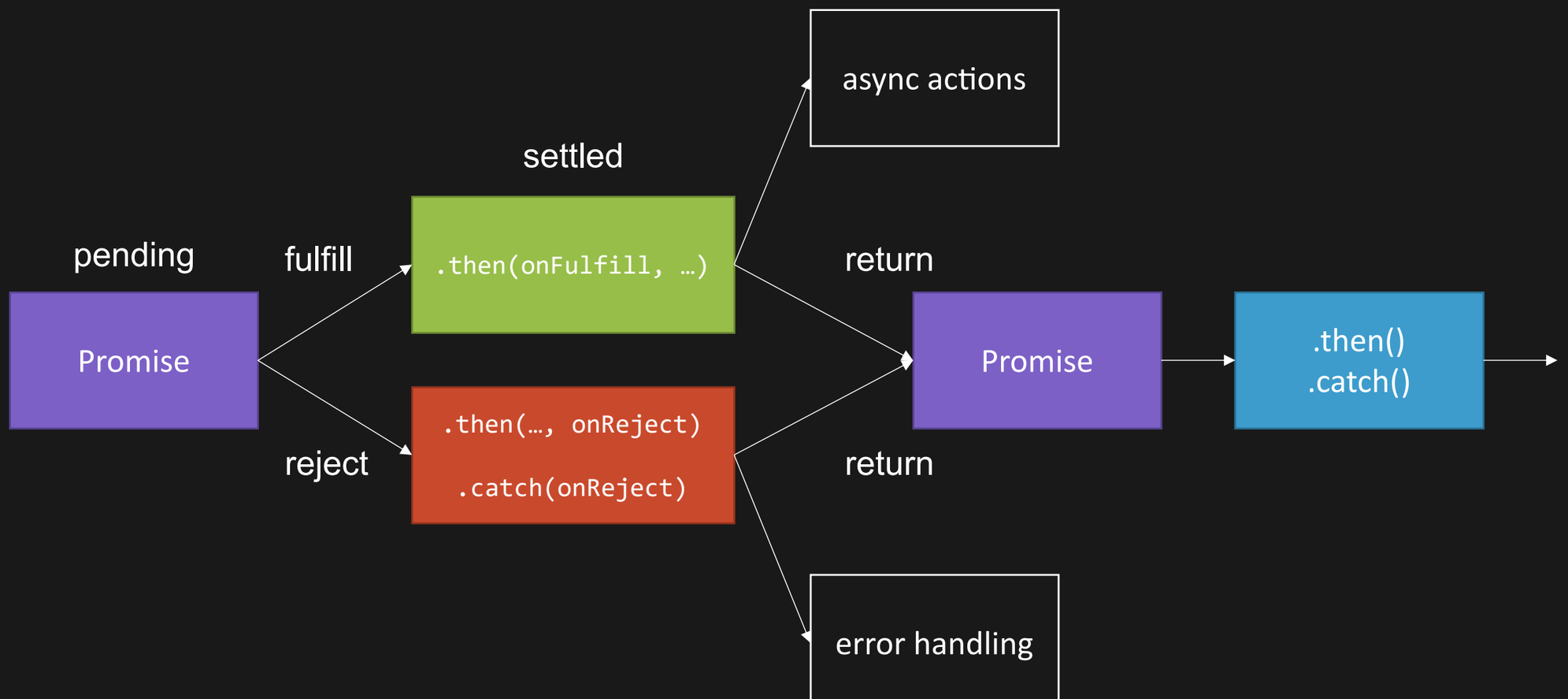
`fulfilled`
: The operation was completed successfully.

`rejected`
: The operation failed.

```
let promise = new Promise((resolve, reject) =>
  ... resolve(data)  // ...
  ... reject(reason) // ...
});
// ...
promise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC)
  .catch(handleRejectedD);
```

A pending promise can either be fulfilled with a value or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's `then` method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

CHAINED PROMISES



PROMISE METHODS

```
1 Promise.all([
2   new Promise(resolve => setTimeout(() => resolve(2), 3000)), // 2
3   new Promise(resolve => setTimeout(() => resolve(4), 2000)), // 4
4   new Promise(resolve => setTimeout(() => resolve(6), 1000)) // 6
5 ]).then(console.log); // 2,4,6 when promises are ready: each promise contributes an array mem
```

STATIC METHODS

- `Promise.all(iterable)`
- `Promise.allSettled(iterable)`
- `Promise.any(iterable)`
- `Promise.race(iterable)`
- `Promise.reject(reason)`
- `Promise.resolve(value)`

INSTANCE METHODS

- `catch()`
- `then()`
- `finally()`

SEQUELIZE: PROMISE-BASED ORM

Sequelize is a promise-based Node.js ORM tool for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server. It features solid transaction support, relations, eager and lazy loading, read replication and more.

```
Product.findAll({ where })
  .catch((reason) => console.error(reason))
  .then((products) => {
    // ... use products
  });
```

```
Product.findOne({ where })
  .catch((reason) => console.error(reason))
  .then((product) => {
    // ... use the product
  });
```

CURRYING

Currying is an advanced technique of working with functions. Currying is the technique of converting a function that takes multiple arguments into a sequence of functions that each takes a single argument. Currying is a transformation of functions that translates a function from callable as `f(a, b, c)` into callable as `f(a)(b)(c)`.

```
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args);
    } else {
      return function (...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  };
}
```

```
function sum(a, b, c) {
  return a + b + c;
}
let currySum = curry(sum);
console.log(currySum(10, 20, 30));
console.log(currySum(10)(20, 30));
console.log(currySum(10)(20)(30));
```

SUPPLANT

Supplant does variable substitution on the string. It scans through the string looking for expressions enclosed in `{{ }}` braces. If an expression is found, use it as a key on the object, and if the key has a string value or number value, it is substituted for the bracket expression and it repeats. This is useful for automatically fixing URLs or for templating HTML.

Based on: Remedial JavaScript

```
function supplant(str, object) {
  return str.replace(
    /\{\{[ ]*([^{} ]*)[ ]*\}\}/g,
    function (a, b) {
      let r = object[b];
      return typeof r === 'string' ||
             typeof r === 'number' ? r : a
    }
  );
}
```

```
const template = 'The distance from {{ from }} to {{ to }} is {{ distance }} km.'
const values   = { from: 'A', to: 'B', distance: 10 };

console.log(supplant(template, values));
// The distance from A to B is 10 km.
```

This slide is intentionally left blank.

Return to Course Page