heig-vd
HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

03 - Towards deployment

Prepare your project for production

heig-vd

Daily menu

- Introducing Javascript promises
- Practice with promises in a real world project
- Learn how to structure client-side and server-side projects
- Deal with browser compatibilities
- Deployment - Heroku and Github Pages

heig-vd

Writing async code is hard

Even if it is easy to understand the notion of callback, it is difficult to orchestrate multiple async operations

Forces

- Writing async code is **not an option** with Javascript.
- It is also becoming increasingly **important in other languages**.
- Writing code with **callbacks quickly leads to problems**: deeply nested “pyramids”, unreadable code, tricky bugs…

Solution (1)

Once upon a time (4 years ago), we had to use libraries such as async.js to orchestrate multiple async operations - parallel(), series(), waterfall(), etc..

```
var async = require('async');

async.waterfall([
    function(callback) {
        callback(null, 'one', 'two');
    },
    function(arg1, arg2, callback) {
        // arg1 now equals 'one' and arg2 now equals 'two'
        callback(null, 'three');
    },
    function(arg1, callback) {
        // arg1 now equals 'three'
        callback(null, 'done');
    }
], function (err, result) {
    // result now equals 'done'
});
```

Solution (1)

Rewrite this pyramid...

```
milkCow( function(err, milk) {
  console.log("I have " + milk + " and can prepare cheese.");
  prepareCheese(milk, function(err, cheese) {
    console.log("I have now " + cheese + " and can sell it.");
    sellCheese(cheese, function(err, money) {
      console.log("Youpi! I have my money.");
    });
  });
});
```

into (cleaner) code

```
async.waterfall([milkCow, prepareCheese, sellCheese], callback);

function callback(err, money) {
    console.log("Youpi! I have my money.");
}
```

Solution (2)

- Promises make dealing with **async** code and **errors** significantly **easier**
- Promises have been proposed as a **standard** way to write async code.
- Initially, there was a specification (Promises/A+) with multiple implementations (bluebird, Q and tens of others).
- Today, Promises have been integrated in ECMAScript. Libraries provide additional features.

Javascript Promises

Let's talk about this

Definition

A `promise` is an object representing the eventual completion or failure of an asynchronous operation, and its resulting value.

Essentially, a promise is an object you **attach callbacks** to...

```
let promise = fetchUser('paulnta');
// Attaching a callback using .then()
promise.then(function(user) {
  // do something with user
});
```

...instead of **passing callbacks** into a function.

```
// Passing a callback
fetchUser('paulnta', function(error, user) {
  // do something with the user
});
```

Creating promises

The **constructor** syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => {
    resolve('Ok, I am done working!');
  }, 1000);
});
```

The promise constructor takes an executor function that lets us

- `resolve(value)` - to indicate that the job finished successfully
- `reject(error)` - to indicate that an error occurred

Creating promises

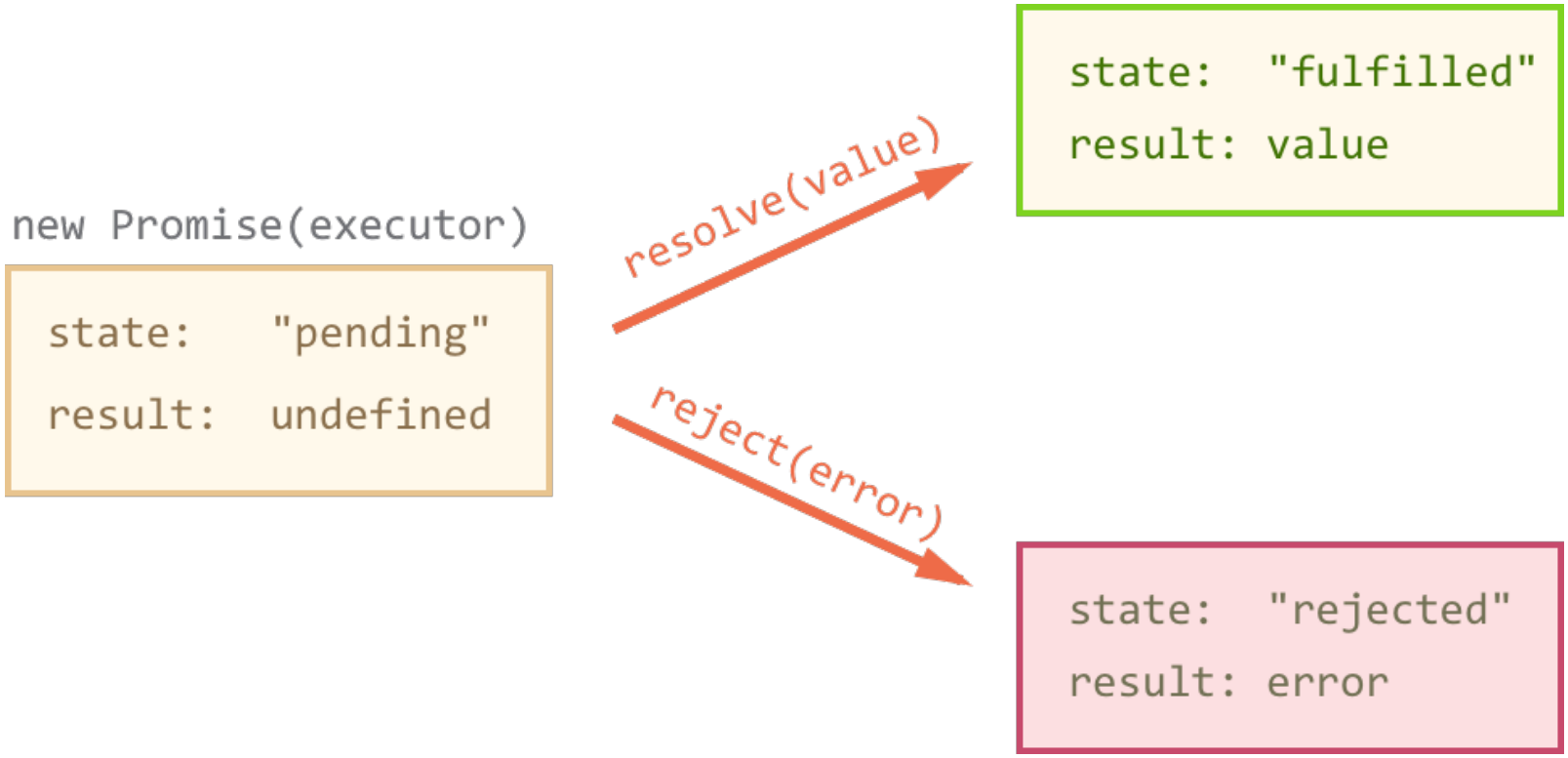
Promises have a `state` property

- `pending` - the operation is not completed yet
- `fulfilled` - the operation is completed
- `rejected` - the operation has failed

Promises have a single `result` property.

- It can be `undefined`
- or the result of the async operation.

Creating promises



Creating promises

```
let promise = new Promise(function(resolve, reject) {
  if (typeof ([] + []) === "string") {
    reject(new Error('Javascript sucks, but I like it!'));
  } else {
    resolve('Thank you!');
  }
});
```

- Only one call to `resolve` or `reject` is taken into account
- `resolve` and reject `accepts` only one argument
- Always call `reject` with error objects
- Immediately calling resolve/reject is totally fine

Ok I have a promise now what ?

```
let promise = functionThatReturnsAPromise();
promise.then(
  function(result) { /* Handle successful result */ },
  function(error) {  /* Handle error */ }
);
```

The first argument of `.then` is a function that:

- runs when the Promise is resolved, and
- receives the result.

The second argument of `.then` is a function that:

- runs when the Promise is rejected, and
- receives the error.

Ok I have a promise now what ?

If you're interested only in **successful** completions, then you can provide only one function argument to `.then`

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(console.log); // logs "done!" after 1 second
```

Ok I have a promise now what ?

If you're interested only in **errors**, then you can use `null` as the first argument: `.then(null, errorHandler)`. Or you can use `.catch(errorHandler)`, which is exactly the same:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

promise.catch(console.log);
// same as promise.then(null, console.log)
// logs "Error: Whoops!" after 1 second
```

Ok I have a promise now what ?

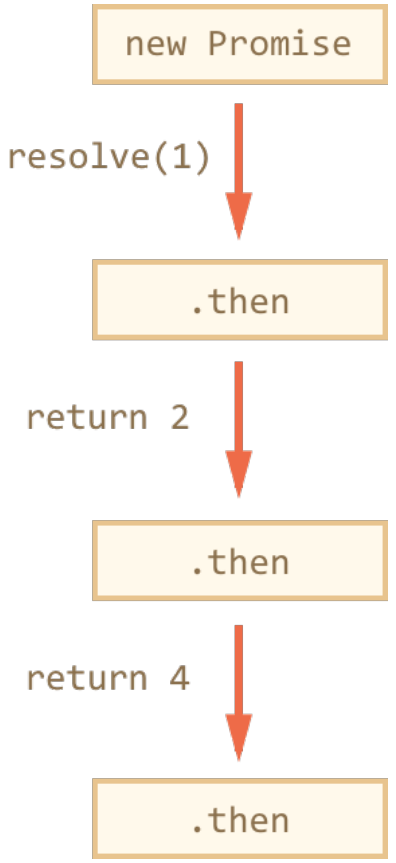
```
let promise = new Promise((resolve) => resolve('done'));

console.log('After promise creation');

promise.then((result) => console.log(`first ${result}`));
promise.then((result) => console.log(`second ${result}`));

// After promise creation
// first done
// second done
```

- Callbacks will never be called before the completion of the current run of the JavaScript event loop.
- Callbacks added with then() even after the success or failure of the asynchronous operation, will be called.
- Multiple callbacks may be added by calling then() several times. Each callback is executed one after another, in the order in which they were inserted.

Promises chaining

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve(1), 1000);
})
  .then((result) => result * 2)
  .then((result) => result * 2)
  .then(console.log);
  // logs 1, in 1 second
```

- Multiple asynchronous tasks can be easily chained.
- a call to `promise.then` returns a promise
- the result of a handler can be a promise
- When a handler returns a value, it becomes the result of that promise



Promises chaining

This code is a messy

```
milkCow( function(err, milk) {
  prepareCheese(milk, function(err, cheese) {
    sellCheese(cheese, function(err, money) {
      console.log("Youpi! I have my money.");
    });
  });
});
```

This code tells a story

```
  milkCow()
    .then(prepareCheese)
    .then(sellCheese)
    .then(money => {
      console.log('Youpi! I have my money.');
    });
```

Real world example

We'll use the fetch method to load the information about a user from the Github API.

Fetch performs a network request and returns a promise. The promise resolves with a response object when the remote server responds with headers, but **before the full response is downloaded**.

```
let promise = fetch(`https://api.github.com/users/paulnta`);
```

To read the full response, we should call a method `response.json()`

```
promise.then(response => response.json());
```

Real world example

```
function getUserAvatar(username) {
  return fetch(`https://api.github.com/users/${username}`)
    .then(response => response.json())
    .then(user => user.avatar_url);
}
```

```
getUserAvatar('paulnta')
  .then(avatar_url => {
    const img = document.createElement('img');
    img.src = avatar_url;
    document.body.append(img);
  })
```

What if `getUserAvatar()` fails ?

> The promise returned by fetch rejects when it's impossible to make a request. 404 or even 500 responses are considered valid

Real world example

```
function getUserAvatar(username) {
  return fetch(`https://api.github.com/users/${username}`)
    .then(response => {
      if (!response.ok) throw new Error('Not found!');
      return response.json();
    })
    .then(user => user.avatar_url);
}
```

```
getUserAvatar('paulnta')
  .then(avatar_url => {
    const img = document.createElement('img');
    img.src = avatar_url;
    document.body.append(img);
  })
  .catch(err => {
    const p = document.createElement('p');
    p.innerText = 'Sorry';
    document.body.append(p);
  })
```

Parallel

You can easily run things in parallel by creating an array of promises

```
const users = ['paulnta', 'edri', 'wasadigi'];
const requests = users.map(getUserAvatar);
```

Then use `Promise.all()` method to wait for all promises to resolve.

```
Promise.all(requests)
  .then(avatars => {
      avatars.forEach(avatar_url => {
        const img = document.createElement('img');
        img.src = avatar_url;
        document.body.append(img);
      })
  })
```

If any of the promises is rejected, `Promise.all` immediately rejects with that error.

References

- Using Fetch
- Promises tutorial

Let's write some code

Github analytics light

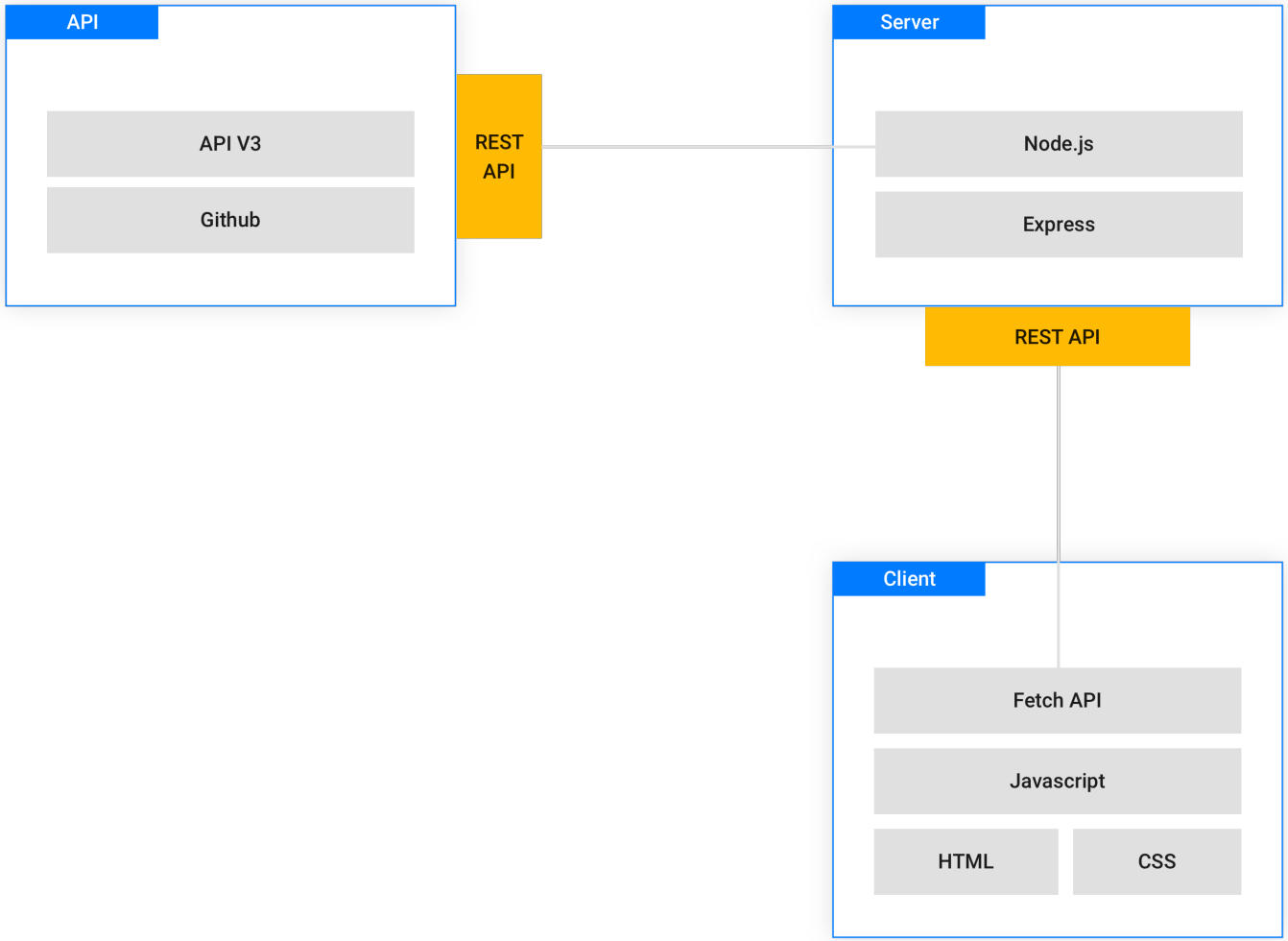
We'll create a really simple Github analytics app. Our app will find users favorite languages based on their public repositories.

The goals are:

- Put into practice **everything we've learned** so far
  - HTML, CSS, Javascript, Node.js, npm, eslint, mocha, etc..
- Practice with **new concepts**
  - Promises and fetch API
  - Javascript DOM manipulation
- Introduce **new concepts**
  - Express - a Node.js framework to create server side applications
  - Build pipelines and deployment

Let's write some code

Github analytics light



Writing next generation Javascript

How do I make sure that my "dialect" of Javascript will work on all browsers?

```
const range = [...Array(5).keys()].map(i => i + 10)
              // [10, 11, 12, 13, 14]
```

Javascript flavours

- We have seen that there are different generations of Javascript (ES5, ES6, etc.).
- You may also have heard about other languages “related” to Javascript: Typescript, CoffeeScript or Flow
- As a developer, you may want to use the last features of the language.
- On the server side, you have control on the runtime environment. On the client side, you don’t.

Solution

- There are tools that transform (transpile) different dialects of Javascript in code that is understood by older engines.
- Today, the most popular is Babel.js.
- It gives you a command line tool, which takes “modern” Javascript files as input and generates “universal” Javascript as output.

Try Babel REPL

Back to basics

- Browsers use different engines (V8, Chakra, SpiderMonkey, JavascriptCore etc..) to translate javascript to machine code.
- Engines are an implementation of the Javascript Language
- Javascript is a general purpose scripting language that conforms to the ECMAScript specification
- ECMAScript is a specification for what a scripting language could look like
- Releasing a new edition of ECMAScript does not mean that all JavaScript engines in existence suddenly have those new features
- Engines incorporate new ECMAScript features incrementally
- ECMAScript 5 (ES5) is widely supported
- Current browsers dont support all the ECMAScript 6 (ES6, ES2015)

What should I do ?

- Define your supported environments. Which browsers (or Node.js versions) your app needs to support
- Write code in the latest version of JavaScript
- If needed, use babel to compile your code down to supported environments

How babel works ?

They are two tasks babel performs when compiling your code we need to understand

Transforming - This tasks transforms the **syntax** of your code. For example Arrow functions `() => {}` needs to be transformed into standard `function() {}` to run on IE 11.

polyfilling - For this task, babel injects polyfills into your code. A polyfill is a code that implement a missing **feature** on a browser. For example `Promise` doesn't exist in IE 11 or Safari 10 and this feature needs to be simulated to work on those browsers.

Babel

There are quite a few tools in the Babel toolchain. Thoses tools (modules) are published as separate npm packages scoped under `@babel`.

- `@babel/core` - Where Babel's core functionality resides. This module is required to use babel.
- `@babel/cli` - Allows you to use babel from the terminal.
- `@babel/polyfill` - contains necessary code to emulate a full ES2015+ environment. *Promise*, *Fetch*, *Array.prototype.includes*, etc...
- `@babel/plugin-transform-xxxx` - Transformations come in the form of plugins, which are small JavaScript programs that instruct Babel on how to transform your code syntax.
- `@babel/preset-xxx` - A preset is just a pre-determined set of plugins.

Set up Babel

To setup babel for command line usage, start by installing dependencies locally

```
$ npm install --save-dev @babel/core @babel/cli @babel/preset-env
$ npm install --save @babel/polyfill
```

Then add a `.babelrc.js` configuration file to specify supported environment. We use a preset called `preset-env` that automatically choose what plugins and polyfills to include based on our supported environment.

```
// .babelrc.js
module.exports = {
  presets: [
    ["@babel/env", {
      // target minimum browser versions
      targets: "> 0.25%, not dead",
      // require needed polyfills
      useBuiltIns: "usage",
    }]
  ]
}
```

Run babel

To compile your code, use the `babel` command line interface. Here we pass an input directory `js` and an output directory `dist` to store the result of the compilation.

```
$ npx babel js --out-dir dist
```

You can also include this command in your `package.json` file as an npm script

```
// package.json
"scripts": {
  "build": "npx babel js --out-dir dist",
  ...
},
```

So know you can run `npm run build` to compile your scripts

> Don't forget to update references any to your scripts in your index.html.

Require modules

Until now, in client-side project, we used script tags to load javascript code.

For babel to work as expected with the previous configuration and to structure your in modules like in Node.js environment, you need a way to `require()` and `exports` modules.

For example you'd like to use `chart.js` npm modules in your app.

```
$ npm install --save chart.js
```

```
// js/app.js
const Chart = require('chart.js');
const myChart = new Chart(ctx, {...});
```

But browsers haven't implemented modules yet. So you'll need an other tool `browserify`.

Require modules

Browsers don't have the `require` method defined, but **Node.js does**. With Browserify you can write code that uses `require` in the same way that you would use it in Node.

Browserify can work side by side with babel (through `babelify`). It also has plugins such as `watchify` to watch for changes in your code and automatically re-build your app.

```
$ npm install --save-dev browserify babelify watchify
```

Running the following command will first run `browserify` to bundle up your modules. Then it'll pass the result to `babelify` which will transform your code and output the result to `dist/app.js`

```
browserify js/app.js -t babelify -o dist/app.js
```

> Replace `browserify` by `watchify` to watch files and re-build automatically.

This is really exhausting !

Right now I want to build features...not spend my entire semester configuring `babel`, `polyfills`, `watchify` or God knows what !

Solution

- Use projects starter templates such as Yeoman
- Use module bundlers such as Webpack, Parcel, Rollup
- Use task runners such as Gulp or Grunt

References

- What's the difference between JavaScript and ECMAScript?
- Compatibility table
- Babel usage guide

How can we deploy our services "to the cloud" ?

How can we deploy our services “to the cloud” ?

- Our app consists of multiple components (services): **front-end** assets, **back-end** API, **crawler**.
- We want to make them publicly available, so we need to “deploy” them somewhere.
- There are many ways to do that, how do we pick one? In our particular case, **money** and **ease of use** are 2 constraints.

Solution

- We can use a combination of “providers” to deploy our different components.
- Serving the client-side assets is very easy with **GitHub Pages**.
- For the back-end API and the crawler, **Heroku** is a PaaS provider that we can use for free.

What is PaaS

- PaaS stands for **Platform as a Service**
- It is one type of “cloud provider”, which allows you to deploy applications - **you don’t worry about the OS, the DB, etc.**
- Other types of “cloud providers” include SaaS (e.g. Google Docs) and IaaS (e.g. Amazon Web Services EC2)

Experiment with Github pages

- Create a repo
- configure Pages
- Push your assets

You'll find a step by step guide in Github pages' home page

Experiment with Heroku

Heroku manages app deployments with Git. It supports many languages - Node.js, Ruby, Java, Scala, etc..

Heroku also integrates well with tools such as Docker.

- Follow instructions in the getting started guide for Node.js
- Explore Add-ons such as mlab and Heroku Scheduler

Github analytics - Todo List

- Validate that we are able to serve our client assets (HTML + CSS + Javascript) with GitHub Pages.
- Follow the Heroku tutorial and learn how to deploy a Node.js application (alternative: look at docker deployment)
- Validate that you are able to deploy your back-end API server to Heroku.
- Experiment with “one-off” dynos and the scheduler add-on.
- validate that you can execute your script on a period basis.