Introducing `hordes`, a module that makes R available from NodeJS.

About

General philosophy

The general idea of `hordes` is that NodeJS is the perfect tool when it comes to HTTP i/o, hence we can leverage the strength of this ecosystem to build Web Services that can serve R results.

For example, if you have a web service that needs authentication, using `hordes` allows to reuse existing NodeJS modules, which are widely used and tested inside the NodeJS ecosystem. Another good example is NodeJS native cluster mode, and external modules like `pm2` which are designed to launch your app in a multicore mode, watch that your app is still running continuously, and relaunch it if one of the process stop (kind of handy for a production application that handle a lot of load).

`hordes` also makes things easier when it comes to mixing various languages in the same API: for example, you can serve standard HTML on an endpoint, and R on others. And of course, it makes it more straightforward to include R inside an existing NodeJS stack.

From the R point of view, the general idea with `hordes` is that every R function call should be stateless. Keeping this idea in mind, you can build a package where functions are to be considered as 'endpoints' which are then called from NodeJS. In other words, there is no "shared-state" between two calls to R—if you want this to happen, you should either register the values inside Node, or use a database as a backend (which should be the preferred solution if you ask me).

But wait…

> … Do I need to learn NodeJS?

Yes! How cool is that!

If unlike me you're not a programming language nerd and don't feel like learning JavaScript and NodeJS, the idea is to make the collaboration between R dev and NodeJS developers/production engineer way easier. R devs can write packages with functions returning data that can be handled by NodeJS, and that way, it will be more straightforward to include R inside a web app that already runs on Node, or to build a new Node app that can use the R functions.

Install

`hordes` can be installed from npm with

```
npm install hordes
```

How to

The `hordes` module contains the following functions:

`library`

`library` behaves as R `library()` function, except that the output is a

JavaScript object with all the functions from the package.

For example, `library("stats")` will return an object with all the functions from `{stats}`. By doing `const stats = library("stats");`, you will have access to all the functions from `{stats}`, for example `stats.lm()`.

> Note that if you want to call functions with dot (for example `as.numeric()`), you should do it using the `[` notation, not the dot one (i.e `base['as.numeric']`, not `base.as.numeric`).

Calling `stats.lm("code")` will launch R, run `stats::lm("code")` and return the output to Node.

Note that every function returns a promise, where R `stderr` rejects the promise and `stdout` resolves it. This point is kind of important to keep in mind if you're building your own package that will be then called through `hordes`.

```
const {library} = require('hordes');
const stats = library(pak = "stats");
stats.lm("Sepal.Length ~ Sepal.Width, data = iris").
    then((e) => console.log(e)).
    catch((err) => console.error(err))
```

```
Call:
stats::lm(formula = Sepal.Length ~ Sepal.Width, data = iris)

Coefficients:
(Intercept)  Sepal.Width
     6.5262      -0.2234
```

As they are promises, you can use them in an async/await pattern or with `then/catch`. The rest of this README will use `async/await`

```
const { library } = require('hordes');
const stats = library("stats");

(async() => {
    try {
        const a = await stats.lm("Sepal.Length ~ Sepal.Width, data = iris")
        console.log(a)
    } catch (e) {
        console.log(e)
    }

    try {
        const a = stats.lm("Sepal.Length ~ Sepal.Width, data = iris")
        const b = stats.lm("Sepal.Length ~ Petal.Width, data = iris")
        const ab = await Promise.all([a, b])
        console.log(ab[0])
        console.log(ab[1])
    } catch (e) {
        console.log(e)
    }
})();
```

```
Call:
stats::lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
```

```
Coefficients:
(Intercept)  Sepal.Width
     6.5262      -0.2234



Call:
stats::lm(formula = Sepal.Length ~ Sepal.Width, data = iris)

Coefficients:
(Intercept)  Sepal.Width
     6.5262      -0.2234



Call:
stats::lm(formula = Sepal.Length ~ Petal.Width, data = iris)

Coefficients:
(Intercept)  Petal.Width
     4.7776       0.8886
```

Values returned by the `hordes` functions, once in NodeJS, are string values matching the `stdout` of `Rscript`.

If you want to exchange data between R and NodeJS, use an interchangeable format (JSON, arrow, base64 for images, raw strings…):

```
const {library} = require('hordes');
const jsonlite = library("jsonlite");
const base = library("base");

(async () => {
    try {
            const a = await jsonlite.toJSON("iris")
            console.log(JSON.parse(a)[0])
        } catch(e){
            console.log(e)
        }
    try {
            const b = await base.cat("21")
            console.log(parseInt(b) * 2)
        } catch(e){
            console.log(e)
        }
}
)();
```

```
{
  'Sepal.Length': 5.1,
  'Sepal.Width': 3.5,
  'Petal.Length': 1.4,
  'Petal.Width': 0.2,
  Species: 'setosa'
}
42
```

mlibrary

`mlibrary` does the same job as `library` except the functions are natively memoized.

```
const {library, mlibrary} = require('hordes');
const base = library("base");
const mbase = mlibrary("base");

(async () => {
    try {
            const a = await base.sample("1:100, 5")
            console.log("a:", a)
            const b = await base.sample("1:100, 5")
            console.log("b:", b)
        } catch(e){
            console.log(e)
        }

    try {
            const a = await mbase.sample("1:100, 5")
            console.log("a:", a)
            const b = await mbase.sample("1:100, 5")
            console.log("b:", b)
        } catch(e){
            console.log(e)
        }
}
)();
```

```
a: [1] 49 13 37 25 91

b: [1]  5 17 68 26 29

a: [1] 96 17  6  4 75

b: [1] 96 17  6  4 75
```

get_hash

When calling `library()` or `mlibrary()`, you can specify a hash, which can be compiled with `get_hash`. This hash is computed from the `DESCRIPTION` of the package called. That way, if ever the `DESCRIPTION` file changes (version update, or stuff like that…), you can get alerted (app won't launch). Just ignore this param if you don't care about that (but you should in a production setting).

```
const { library, get_hash } = require('hordes');
console.log(get_hash("golem"))
```

```
'fdfe0166629045e6ae8f7ada9d9ca821742e8135efec62bc2226cf0811f44ef3'
```

Then if you call `library()` with another hash, the app will fail.

```
var golem = library("golem", hash = "blabla")

            throw new Error("Hash from DESCRIPTION doesn't match specified
hash.")

var golem = library("golem", hash = 'e2167f289a708b2cd3b774dd9d041
b9e4b6d75584b9421185eb8d80ca8af4d8a')
Object.keys(golem).length
```

`waiter`

You can launch an R process that streams data and wait for a specific output in the stdout.

The promise resolves with and `{proc, raw_output}`: `proc` is the process object created by Node, `raw_output` is the output buffer, that can be turned to string with `.toString()`.

A streaming process here is considered in a lose sense: what we mean here is anything that prints various elements to the console. For example, when you create a new application using the `{golem}` package, the app is ready once this last line is printed to the console. This is exactly what `waiter` does, it waits for this last line to be printed to the R stdout before resolving.

```
> golem::create_golem('pouet')
-- Checking package name ---------------------------------------------------
v Valid package name
-- Creating dir ------------------------------------------------------------
v Created package directory
-- Copying package skeleton ------------------------------------------------
v Copied app skeleton
-- Setting the default config ----------------------------------------------
v Configured app
-- Done --------------------------------------------------------------------
A new golem named pouet was created at /private/tmp/pouet .
To continue working on your app, start editing the 01_start.R file.
```

```
const { waiter } = require("hordes")
const express = require('express');
const app = express();

app.get('/creategolem', async(req, res) => {
    try {
        await waiter("golem::create_golem('pouet')", solve_on = "To continue
working on your app");
        res.send("Created ")
    } catch (e) {
        console.log(e)
        res.status(500).send("Error creating the golem project")
    }
})

app.listen(2811, function() {
    console.log('Example app listening on port 2811!')
})
```

-> http://localhost:2811/creategolem

Changing the process that runs R

By default, the R code is launched by `RScript`, but you can specify another (for example if you need another version of R):

```
const { library } = require('hordes');
const base = library("base", hash = null, process = '/usr/local/bin/RScript');

(async() => {
    try {
        const a = await base.sample("1:100, 5")
        console.log("a:", a)
    } catch (e) {
        console.log(e)
    }
})();
```

Examples

Simple example

```
const { library } = require('hordes');
const dplyr = library("dplyr");
const stats = library("stats");

(async() => {
        try {
            const sample = await dplyr.sample_n("iris, 5")
            console.log(sample)
        } catch (e) {
            console.log(e)
        }

        try {
            const pull = await dplyr.pull("airquality, Month")
            console.log(pull)
        } catch (e) {
            console.log(e)
        }

        try {
            const lm = await stats.lm("Sepal.Length ~ Sepal.Width, data = iris")
            console.log(lm)
        } catch (e) {
            console.log(e)
        }
    }

)();
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1          5.7         3.8          1.7         0.3     setosa
2          6.7         2.5          5.8         1.8  virginica
3          6.9         3.1          5.1         2.3  virginica
4          6.4         2.9          4.3         1.3 versicolor
5          5.1         3.3          1.7         0.5     setosa

  [1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6
 [38] 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7 7 7 7
 [75] 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
[112] 8 8 8 8 8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
[149] 9 9 9 9 9
```

```
Call:
stats::lm(formula = Sepal.Length ~ Sepal.Width, data = iris)

Coefficients:
(Intercept)  Sepal.Width
     6.5262      -0.2234
```

API using express

```
const express = require('express');
const { library } = require('hordes');
const app = express();
const stats = library("stats");

app.get('/lm', async(req, res) => {
    try {
        const output = await stats.lm(`${req.query.left} ~ ${req.query.right}`)
        res.send('


' + output + '

') } catch (e) { res.status(500).send(e) } }) app.get('/rnorm', async(req, res) => { try { const output = await stats.rnorm(req.query.left) res.send('


' + output + '') } catch (e) { res.status(500).send(e) } }) app.listen(2811, function() { console.log('Example app listening on port 2811!') })
```

->
http://localhost:2811/lm?left=iris$Sepal.Length&right=iris$Petal.Length

-> http://localhost:2811/rnorm?left=10

golem Creator

```
const { waiter } = require("hordes")
const express = require('express');
const app = express();

app.get('/creategolem', async(req, res) => {
    try {
        await waiter(`golem::create_golem('${req.query.name}')`, solve_on = "To
continue working on your app");
        res.send("Created ")
    } catch (e) {
        console.log(e)
        res.status(500).send("Error creating the golem project")
    }
})

app.listen(2811, function() {
    console.log('Example app listening on port 2811!')
})
```

-> http://localhost:2811/creategolem?name=coucou