

## Validación y manejo de errores

La "validación" es el acto de confirmar si los datos que necesitamos están llegando (a través de la API) a nuestra API. Al ser logica del negocio, va en el controlador.

Se valida a través de condicionales. o .exec()

```
async createUser (req, res) {
  const data = req.body
  if (data.name & & data.age) {
    const user = await this.userService.addUser(data)
    res.send('usuario creado')
  }
}
```

lógica del negocio : todas las decisiones de diseño / de arquitectura para el funcionamiento de la API

**EJE 01** : ejecutar la query de acuerdo a los parámetros q' ponemos.

Se puede hacer manejo de error a través de un **custom**:

```
deleteUser () {
  const query = User.deleteOne ({ name: "Bel" }, function (err) {
    if (err) console.log(err)
    console.log("Successful deletion", { query })
    return query
  })
}
```

Esto reemplaza a .exec() en el manejo de errores custom.

El **manejo de errores** es la acción de informar a quien los sufre (o trae) de lo que pasó. Así si la acción que intentó realizar fue un error o un éxito. También se realiza en el controlador.

```
async createUser (req, res) {  
    const data = req.body;  
    if (data.name & & data.age) {  
        const user = await this.userService.addUser(data);  
        res.send("usuario creado");  
    } else {  
        res.status(400).send("falta información");  
    }  
}
```

} sin manejo de errores

/ **Validación** → todo lo q. podemos controlar del lado del usuario

/ **manejo de errores** → cuando algo sale mal del otro lado  
↓ q' hacemos para inf. al usuario?

\* **try / catch** : bloq de código condicional q' primero intenta realizar 1 acción. Si devuelve 1 error, lo atrapa

```
try { nonExistentFunction(); } catch (error) { console.log(error); }
```

```
async createUser (req, res) { const data = req.body;  
if (body.name & & body.age) {  
    try { const user = await this.userService.addUser(data); res.send("usuario creado"); }  
    catch (e) { console.log(e); res.status(500).send("Hubo un error en la ejecución"); }  
} else { res.status(400).send("falta información obligatoria"); }  
}
```

NOTA

## Paginado

El paginado es la acción de devolver una lista dividida en partes o páginas. Esto nos permite realizar queries más pequeñas y no sobrecargar la base de datos. Se hace en el service que se interactúa c/ la db.

Se usan dos métodos:

\* **skip()**: sirve para saltar rdes a la hora de devolver los registros.

```
getUser(data) {  
    const query = User.find().skip(3).exec(); } mos trae todo los  
    return query  
} registros a partir del  
lindo xq salte los  
primeros tres.
```

\* **limit()**: sirve para devolver una q máxima de rdes

```
getUser(data) {  
    const query = User.find().limit(10).exec(); } mos trae 10  
    return query  
} registros diez.
```

Tanto para skip como limit, la dato debe ser un entero del usuario. (Número del nº de query)

Para armar un paginado necesitamos un límite y página y un valor de offset (Casi q tenemos q ignorar segun el nº de página en la q estamos).

El offset se calcula :  $\text{limit} \times (\text{Page Number} - 1)$

(1)

(2)

Para un  $\text{limit}$  <sup>limit</sup> (1) de 20 entradas = (1)  $20 \times (3 - 1) = 40$

• Se donde viene el dato de pageNumber y de limit?  
Este dato viene de req.query. (? clave = valor)  
la envia el FE

req.query → Propiedad del obj req que nos permite acceder  
a info en la url.

localhost:3000/getfirms?limit=20 & page=20

Ej. Desafio Services : (1) get y post (5) (7)  
(2) (6)  
(3)

## Combo controllers y modelos

- ① Crear nuevo proyecto
- ② " nueva db"
- ③ Conectar a la db desde www
- ④ Llenarla la db c/ datos mediante robust
- ⑤ Crear 2 modelos:

```
const mongoose = require("mongoose")
```

```
const userSchema = mongoose.schema({
```

```
name: {
```

```
type: string,
```

```
required: true
```

```
}
```

```
})
```

Module.exports = mongoose.model("User", userSchema)

- ⑥ Creamos las colecciones y las llenamos
- ⑦ Hacemos un get (route dynamic):

```
router.get("/:id", function(req, res) {
  res.send("User")
})
```

```
router.get("/user/:name", function(req, res, next) {
```

```
// cosas
```

```
})
```

- ⑧ Creamos controllers:

```
class UserController {
  constructor (userService) {
    this.userService = userService
  }
  getUserByName (req, res) {
    // Llamar a un metodo del servicio
    res.send ('OK');
  }
}
```

module.exports = UserController

(2) Creamos el service :

```
const User = require ('./model/UserModel.js')
```

```
class UserService {
```

```
  getUsenByName (name) {
    const query = User.findOne ({name: name}).exec();
    return query;
  }
}
```

module.exports = UserService

(10) en el controller , llamamos al service

- req al servicio

- llamar al metodo del servicio :

```
const user = this.userService.getUserByName (name)
```

- agregar async / await → llamada al metodo sync

↓ nombre del metodo controller

→ lo tomamos de los paráms

```
console.log (user)
```

NOTA

## ⑪ instanciar service y controlar en los routers

- UserService = require './service'
- UserController = " " UserController
- UserInstance = new UserController(new UserService())

~~scribble~~

- en router.get:

UserInstance.getUserByName (req, res)

2:04:52

¿ Cómo se hace la conexión a la collection desde el código? Para saber en qué collection buscar específicamente

Nosotros hacemos la conexión a toda la base de datos y lo que hace mongoose es la conexión a la collection cuando definimos los modelos. Al hacerlo en singular, mongoose sabe qué tiene que buscarnos en la collection (en plural). Al hacer un save() mediante un post, si no existe creará la collection, le ~~base~~ crea mongoose automáticamente

## ⑫ en el controller, modelos de dato

res.json (user);

const data = {

name: user.name

favDrink: user.favDrink }



→ res.json (data)

NOTA

(13) Entregamos a tres informe desde la colección de Juncos, haciendo 1 query en esa colección a través del DrinkService

const **Drink** = require 'models/Drink'

class **DrinkService** {

**getDrinkBy<sup>Name</sup>** (<sup>name</sup>) {

    const query = Drink.findOne ({name: name}); exec (),

    return query

}

module.exports = **DrinkService**;

(14) El pedido de info se hace en el - lugar donde se hace la Request: el rute

- requestor **DrinkService**

- en **Userinstance** : new **UserController** (new **UserService** (), new **DrinkService** ());

(15) En el controller, poner **DrinkService** como variable de clase en el constructor

constructor (**userService**, **drinkService**) {

this.userService = userService;

this.drinkService = drinkService;

}

(16) Accedemos a la inf. de la query de **DrinkService** desde el controller

NOTA

- ~ C'est dmnu = Swrit their drink service getDrinkByName (User)  
Favolous
- y en el resultado de dato se regresa en la Not. "man"  
man : dmnu.mainIngredient

## Manejo de archivos con multer

Algoritmo

Multer es una librería que permite el envío de archivos a nuestro servidor

comando: `npm install multer`

### ① Configurando multer en index.js (en la carpeta del proyecto)

```
const multer = require ("multer")
```

```
const storage = multer.diskStorage ({
```

destination: "uploads",

```
destination: function (req, file, cb) {  
    cb (null, "uploads")  
},
```

```
filename: function (req, file, cb) {
```

```
    cb (null, file.fieldname + "-" + Date.now () + ".png")  
},
```

```
});
```

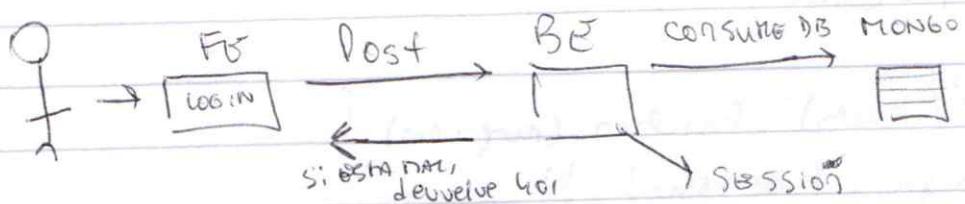
```
const upload = multer ({ storage: storage });
```

## Passport - Business Session

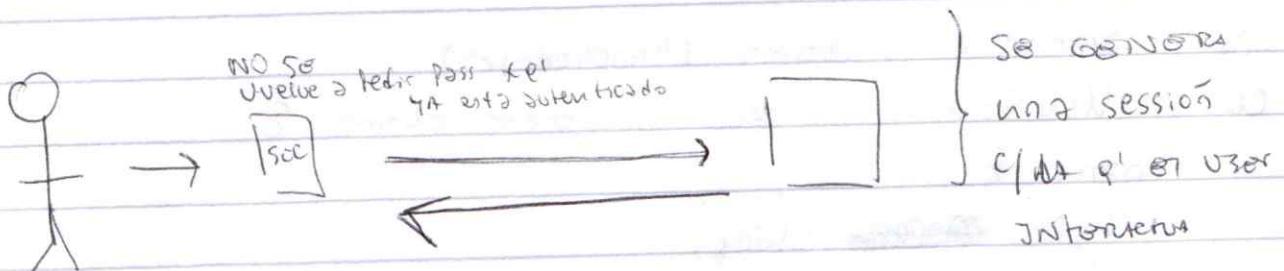
Es una librería para autenticar usuarios. Funciona como un middleware.

**cookies:** → lugar para guardar información de sesión.

**autenticación:** → relacionado c/ roles / perfiles



Se hace con un **post** xq viene "body" (get no lo tiene + lo q' hay q' mandar inf a través de la url y eso está mal) q' es + seguro q' mandar inf. sensible



45.54

Autenticación

Loguearse

Validarse

Las claves se hashean pero los controlan si es la persona. Se comparam los hashes: el de la db y el q' esté generando la persona al loguearse.

NOTA

① Instalar las 2 librerías: (A) passport (B) passport-local

→ para login

c/ email y password

② creamos un archivo ~~local.js~~ en la raíz del proyecto (Passport.js)  
y luego lo requerimos en app.js, arriba de express:

```
const passportConfig = require("./passport");
```

③ ruta al login y al verify (session):

```
(a) router.post("/login", function (req, res) {  
    return res.json({ ok: true }); })
```

```
(b) router.get("/verify", function (req, res) {  
    return res.json({ ok: true }); })
```

④ creamos el model

```
const mongoose = require("mongoose")
```

```
const UserSchema = new mongoose.Schema({
```

```
    name: {
```

```
        type: object string,
```

```
        required: true,
```

```
    }
```

```
    password: {
```

```
        type: string,
```

```
        required: true,
```

```
    }
```

```
module.exports = mongoose.model("User", UserSchema);
```

## - Crear el Service

```
const User = require ('.. /model/User')
```

```
class UserService {
```

```
    getByName (Name) {
```

```
        const query = User.findOne ({ name }) . exec () ;
```

```
        return query;
```

```
}
```

en un Obj.  
lo toma como key/value

```
module.exports = UserService;
```

## - creamos datos en ROBOT

```
{ "name": "Coty",  
  "password": "clave segura1234"  
}
```

## - Vamos a las routes y requerimos Passport como middleware

```
const passport = require ('passport')
```

- en route.post ('/login') escribimos:

```
router.post ('/login', passport.authenticate ('local'), (req, res) => {
```

passport.authenticate ejecuta una función q' utiliza la estrategia local q' mega mediante post y validarla q' ve si tienen un db

• En passport.js escribirnos la estrategia Local :

- Frecuentes Passport : const Passport = require("passport")
- " " la LocalStrategy : const LocalStrategy = require("passport-local").Strategy;

passport.use (new LocalStrategy ({

usernameField: "name",

passwordField: "password"

}, {username, password, cb}) => {

• Creamos una instancia de nuestro socio y llamamos al metodo

const UserService = require("./services/UserService")

const UserInstance = new UserService();

const userDatas = await UserInstance.getByName(username);

• creamos la logica

✓ Si el usuario es #

if (!userDatas) {

return cb(null, false);}

✓ Si el password es #

if (userDatas.password != password) {

return cb(null, false);}

✓ Si esta bien

return cb(null, userDatas)

• En app.js meter alle los routes :

app.use (Passport.initialize());

app.use (Passport.session());

NOTA

- en passport.js:

```
Passport.serializeUser ((user, cb) => {  
  cb (null, user.username);});
```

```
Passport.deserializeUser (async (name, cb) => {  
  const data = await UserInstance.getByName (name);  
  cb (null, data);});
```

- 3b Con la información del login, Passport crea un obj. llamado **req.user**

- instalamos la librería "**express-session**"

- requerimos la lib en **app.js**:

```
const session = require ("express-session");
```

- creamos la sesión:

```
const sessionMiddleware = session ({  
  name: "auth-sit",  
  secret: "phelybrh-s3cr3t", → es lo que hace para encriptar  
  saveUninitialized: false,  
  resave: false  
})
```

- esto lo ponemos arriba de **passport.initialize**:

```
app.use (sessionMiddleware);
```

- creamos la route:

```
router.get('/verify', function (req, res, next) {  
  return res.json({ user: user });  
});
```

funcion que devuelva el resultado de la validación

NOTA

un node → hace todos los proyectos colgados

HOUAN  
FECHA

### en passport.js:

```
Passport.serializeUser ((user, cb) => {  
  cb (null, user.username);});
```

```
passport.deserializeUser (async (name, cb) => {  
  const data = await UserInstance.getByName (name);  
  cb (null, data);});
```

3b Con la información del login, passport crea un obj. llamado **req.user**

- instalamos la librería "**express-session**"

- requerimos la lib en **app.js**:

```
const session = require ("express-session");
```

- creamos la sesión:

```
const sessionMiddleware = session ({  
  name: "auth-test",  
  secret: "phelybrh-s3cr3t", → es la palabra clave secreta  
  saveUninitialized: false,  
  resave: false  
})
```

- esto lo lleva arriba de **passport.initialize**:

```
app.use (sessionMiddleware);
```

NOTA

— creamos la route:

```
router.get('/verify', function (req, res, next) {  
  return res.json({ user: user });  
});
```

— creamos "User" y "Verify" en el modelo

— "Verify" tiene un campo

— "Verify" tiene una relación con "User"

— "Verify" tiene un campo

NOTA

## Bcrypt :

Es una función de cifrado para hashear contraseñas y así evitar guardarlos en texto plano.

Se instala con : `npm i bcrypt`

Se hace a través de 2 métodos: (A) hash

(B) compare

(A) Método hash : Recibe como parámetro el password en texto plano y un nº de "saltrounds" para complejizar el cifrado (default: 5 - b64: 10)

`bcrypt.hash(data.password, 10).then(hash => {  
 // inmestras acciones  
});`

(B) Método compare : Recibe como parámetro nuestro password en texto plano y el hash. Realiza la comparación y nos avisa si son iguales o no. (true/false)

`bcrypt.compare(password, hash)`

## A) Método hash al crear / modificar un usuario

```
const User = require("../models/UserModel");
const bcrypt = require("bcrypt");

class UserService {
  create(data) {
    bcrypt.hash(data.password, 10).then((hash) => {
      data.password = hash;
      const newUser = new User(data);
      return newUser.save();
    });
  }
}

module.exports = UserService;
```

## B) Método compare Al logear un usuario

```
bcrypt.hash(wanted.password, 10).then((hash) =>
  bcrypt.compare(password, hash));
return cb(null, false);
});
```

## Implementando Bcrypt

- ① Crear el controller para crear el usuario:

```
class UserController {
  constructor (UserService) {
    this.UserService = UserService
  }

  23/12 CreateUser (req, res) {
    const { body } = req
    const name = body.name.toLowerCase ()
    if (body & & body.name & & body.password) {
      } else { return sendStatus (400); } }
  }
```

module.exports = UserController

\* Hacemos un try/catch

```
try { this.UserService.createUser ({ ... body, name }) }  
catch (e) { console.log (e) }
```

res.sendStatus(200).json(user)

## ② Creamos el servicio.

```
class class CreateUser (data) {  
    bcrypt.hash(data.password, 10).then (hash => {  
        data.password = hash;  
        const newUser = new User(data);  
        return newUser.save();  
    });  
}
```

module.exports = UserService;

NO OLVIDAR : Requerir el model & le concatenar

## ③ Vamos a las rutas e instanciamos.

- Requerimos Controller y service
- creamos UserCustomer con controller y service
- creamos la route:

```
router.post('/create', function (req, res) {  
    Userinstance.createUser (req, res)  
})
```

## Comprobando hashes

- (1) respuesta bcrypt en passport.js
- (2) en el cheques de Userdata:

```
const compare = await bcrypt.compare(password, userdatas.password)  
console.log(compare);
```

- (3) en el segundo if :

```
if (!compare) { código del 2º if }
```

## Deployar código Ali (2:n)

## Custom middleware

Middleware es una función q' se ejecuta entre la req y la logica q' dispone una request

Cualq Funcionalidad genérica de nuestra aplicación, ce podemos guardar en la carpeta "utils"

Para utilizarla no hay q' olvidarse de : ① module.exports  
 ② responder en donde le queremos

function CustomMiddleware (req, res, next) {

u el body de esa función

}

module.exports = CustomMiddleware;

- ① creamos la carpeta "utils"
- ② creamos el file checkAdmin.js
- ③ En checkAdmin.js:

```
function checkAdmin (req, res, next) {
  if (req.user) {
    if (req.user.admin) { console.log("sos admin"); }
    else { console.log("no es admin"); res.sendstatus(403); }
  } else { console.log("no hay user logueado"); res.sendstatus(401); }
  * next()
}
```

- 4 La requierimos en router y le usuarios antes de la req

## PROMESAS

Son objetos de JS. Se puede instanciar. Tiene un constructor q' recibe una fun. q' 2 parámetros: **resolve**, **reject**. Indican q' algo se está ejecutando.

~~const promise = new Promise(function(resolve, reject) {~~

// hace algo

~~if ( todo sale bien ) { resolve("esto es lo q' recibimos en un try") }~~  
 ~~else { reject("error" ("esto es lo q' recibimos en un catch")) }~~  
};

Axios.get(url) → esto es una promesa

El try/catch es la forma + moderna de resolver una promesa pero también se usa **.then()** / **.catch()**

Axios.get(url)

```
.then(user => {
  // si la promesa se resuelve
  console.log(`User: ${user.data}`);
  res.sendStatus(200);
})
```

// si la promesa se rechaza

```
.catch(error => {
  res.status(404).send("User no encontrado")});
```

Tienen sus propios métodos

.then()

.catch()

.all() → q' se necesita un array de promises; esto es  
a q'se resuelven todas para realizar una acción

collection: Users  
Model: User

HOJA N°

FECHA

Models → represents collections in Mongo (User)

Schemas → defines the structure of our data objects

Mongoose → adds a layer of methods to easily save, edit, retrieve and delete data from MongoDB

/ get request

### • SERVICE

```
getUsers () {  
  const query = User.find();  
  return query;  
}
```

### • CONTROLLER

```
constructor (userService) {  
  this.userService = userService  
}
```

```
async getUsers (req, res) {  
  const users = await this.userService.getUsers();  
  return res.json (users)  
}
```

### • ROUTES

```
route.get ('/').function (req, res, next) {  
  UserInstance.getUsers (req, res);  
};
```

Se crea una instancia

c/ el servicio como PARámetro  
en el controlador

## Métodos de Mongoose

getUser()

✓ NombreModel.find().exec(), → devuelve un array c/  
rdenados

getUser(id)

✓ NombreModel.findOne(filter).exec(), → devuelve un obj en  
({ nombrefiltro }).exec(), base de datos

getUser(id, data)

✓ NombreModel.findOneAndUpdate(Filter, DATA).exec(), → devuelve  
({ \_id: id }, data) un obj en base  
al filtro y lo modifica  
en base a data

deleteUser()

✓ NombreModel.deleteOne({ name: 'Bel' }), function(err) {

if (err) console.log(err);

console.log('Successful Deletion');

});

return query

→ borra una entrada

q controla q el filtro

addUser(data) {

✓ const newUser = new User(data); → crea la entrada  
en el db.

return newUser.save()

}

- Model : q' p'los va a tener nuestro modelado de datos  
↓  
y q' tipo de datos  
es requerido en el service

- Se hace a traves de un schema
- El nombre debe ser : ✓ singular  
✓ Capitalizado } User  
✓ de tipo string

- Service: donde viven las func q' interactuan con la db

↓  
es requerido en el controllers y los routes

- es una clase (para poder instanciarla)
- tiene metodos (el codigo del metodo va a cí)

✓ En el controller, como variable de clase :

```
class UserController {  
    constructor (userService) {  
        this.userService = userService  
    }  
}
```

```
async getUsers (req, res) {  
    const users = await this.userService.getUsers()  
    return res.json(users)  
}  
}
```

✓ en las routes ; instanciamos el service

```
const UserController = require("./../controllers/UserController");
const UserService = require("./../services/UserService");

const UserInstance = new UserController(new UserService());
```

- Deployar una API → usamos Heroku

— o —

Custom API Middleware → sirve para asignar Roles

Cualquier función q' se ejecuta entre el request y el login  
q' dispara esa respuesta

• utils : es una carpeta donde podemos guardar funciones de funcionalidad genérica.

• sintaxis :

- lo requerimos en el archivo q' queremos utilizar
- lo agregamos en la ruta:

```
router.get("/comiddle", middleware, function (req, res, next) {
  res.send("primero corre el middleware y despues esto");
});
```

• Para checar si alguien es Admin o no:

- ① en utils creamos ~~checkAdmin.js~~ checkAdmin.js
- ② en el model, requerimos lo q' es isAdmin: { type: Boolean }
- ③ en la db, creamos un user q' name # password y isAdmin: true
- ④ en checkAdmin.js:

```
function: checkAdmin (req, res, next) {
  const log = console.log(`req: ${req} res: ${res}`);
  if (req.user) {
    if (req.user.isAdmin) {
      log(`A`);
      if (B) {
        console.log(`el user es admin`); } else { console.log(`no es admin`); }
      else { console.log(`no has usado log`); }
    }
  }
}
```

(h)

A) next()

B) res.status(403).send("no sos admin");

C) res.status(401).send("no estas logueado");

D) en router → next() es una función q' le dice al código q' execute lo q' sigue

```
router.get("/onlyadmins", checkAdmin, (req, res, next) => {  
  res.send("solo me acceden admins");  
});
```

## PROMESAS

1) Crear un nuevo proyecto: node -g et

npx ~~create~~ express generator

Las promesas son objetos de JS, se pueden instanciar

```
const promise = new Promise(function(resolve, reject) {
```

// haceremos algo

```
    if (sale bien) { resolve("esto es lo q' recibiras en un try"); }  
    else { reject(Error("esto es lo q' recibiras en un catch")); }  
});
```

Otra versión moderna en un try/catch → mejor forma

Otra forma:

NOTA

EXERCICIOS

```

    .get ('url / $ { req.params.name } ')
    .then (user => {
        // si se resuelve la promesa
        console.log (user.data); res.sendStatus (200);
    })
    .catch (error => {
        // si no se resuelve la promesa
        res.sendStatus (500);
    });
}

```

## (2) MÉTODOS

- `Promise.then()` → recibe como un parámetro una función que ejecuta si la promesa se resuelve
- `Promise.catch()` → recibe como un parámetro una función que ejecuta si la promesa se rechaza
- `Promise.all()` → recibe como parámetro un array de promises y estas se resuelven todas para realizar 1 acción

## (3) const EXERCICIOS = new EXERCICIOS

```

router.get ('/user/:name', function (req, res) {
    const { name } = req.params;
    EXERCICIOS.get ({ url: `https://jsonplaceholder.typicode.com/users/${name}` }), res.send ('no se encue');
    res.send ('no se encue');
})

```

pokemon

```
• then (pokemon) => {  
    console.log(pokemon.data); }  
• catch (error => {  
    console.log(error); })
```

~ ~

① try/catch → async (req, res) => {

try {

```
const user = await axios.get(`api.github/user/${req.params.name}`);  
console.log(user.data);
```

const userData = {

name: user.data.name,  
following: user.data.following  
};

res.send(userData)

json

modelado de  
data

} catch (e) {

console.log(e);

res.sendStatus(400);

? pokemon = nombre de

② ? clave → query.params } ? = variable  
? = nombres

console.log(req.query)

Para pedir el pokemón favor de un user

```
const pokemon = await axios.get(`api.pokemon/pokemon/${  
    req.query.pokemon}`)
```

Agregamos al modelado:

favIcon = pokemon.data.sprites.front\_default;

NOTA

- (B) rawlet, piexachn, genor → se entiende q' quiere mandar un array

```
const paciones = req.query.paciones.split(',');
console.log(paciones)
```



- Para iterar sobre el array

```
const pacionesData = paciones.map(pacion =>
  {
    const data = await axios.get(`api/paciones/
    ${pacion}/$${pacion}`); *
    return data;
});
```

- Devuelve <pending>: porque no llegan a resolverse
- Se usa promise.all:

```
const finalPacionesData = await promise.all(pacionesData)
```

↳ devuelve otra promise{<pending>}

(2. 38 : lo vuelto del break)

- para mostrar el array de promesas resueltas:  
En el modelado: favelores: finalPacionesData

\* return data.data.sprites.front.default