CREACIÓN DE UNA API

TIEMPO ESTIMADO: 150 MINUTOS

DESCRIPCIÓN

- *Creación de una API REST estándar y completa que pueda clonarse para distintos proyectos*

OBJETIVOS

- Conocer el objeto Router de express
- Conocer arquitecturas MVC
- Familiarizarse con base de datos no relacional
- Comprender las ventajas en JavaScript de MongoDB frente a bases de datos relacionales
- Familiarizarse con el uso de contenedores (docker)

PASOS PREVIOS

- Entender como funciona un servidor express (proyecto anterior)
- Tener claro que es una API y una arquitectura API REST

PRIMEROS PASOS

- Fork del repositorio en GitHub
- Clonar tu repositorio
- Inicializar proyecto con npm init
- Instalar y configurar eslint extendiendo de standard:

```
npm i -D eslint@5.4.0
node_modules/.bin/eslint --init
```

- Personalizar eslint utilizando el fichero *.eslintrc.json*

INSTALAR PAQUETES Y CONFIGURAR NPM START

- Instalamos express

```
npm i -S express@4.16.3
```

- Configuramos package.json para que nuestro servidor arranque mediante *npm start*

```
"scripts": {
  "start": "node app/server.js"
},
```

POSTMAN

- Utilizaremos Postman para testear nuestra API
  - Independiente de que hagamos tests por otro lado
- Es una herramienta muy extendida

COMPROBACIÓN API INICIAL

- Arranca la aplicación y comprueba su funcionamiento mediante Postman

```
var express = require('express') //llamamos a Express
var app = express()

var port = process.env.PORT || 8080  // establecemos nuestro p

app.get('/', (req, res) => {
  res.json({ mensaje: '¡Hola Mundo!' })
})

app.get('/cervezas', (req, res) => {
  res.json({ mensaje: '¡A beber cerveza!' })
})

// iniciamos nuestro servidor
app.listen(port)
```

IMPLEMENTACIÓN API

AÑADIR RUTAS

- Añade la ruta **POST /cervezas** con respuesta:

```
{ "mensaje": "Cerveza guardada" }
```

- Añade la ruta **DELETE /cervezas** con respuesta:

```
{ "mensaje": "Cerveza borrada" }
```

- Muestra el mensaje *API escuchando en el puerto 8080* **justo cuando se levante el puerto**
- Comprueba funcionamiento mediante Postman
- Podemos utilizar la extensión ExpressSnippet para autocompletado.

COMMIT CON LAS NUEVAS RUTAS

- Hagamos un commit del repositorio, sin la carpeta node_modules

```
git status
git add -A app/server.js
git commit -m "Añadido post y delete, arreglado error"
git push
```

- Debemos hacer nuevas instantáneas (commits) en cada paso
  - Aquí no se documentarán por brevedad
  - Aquí hemos mezclado un parche con una funcionalidad nueva. ¡No es lo correcto!

NODEMON

- Es un wrapper de node, para reiniciar nuestro API Server cada vez que detecte modificaciones.

```
npm i -D nodemon
```

- Cada vez que ejecutemos **npm start** ejecutaremos nodemon en vez de node. Habrá que cambiar el script en el fichero *package.json*:

```
"start": "nodemon app/server.js"
```

USO DE ENRUTADORES

- Normalmente una API:
  - Tiene varios recursos (cada uno con múltiples endpoints)
  - Sufre modificaciones -> mantener versiones
- Asociamos enrutadores a la app en vez de rutas:
  - Cada enrutador se asocia a un recurso y a un fichero específico.
  - Se pueden anidar enrutadores (*router.use*)

- El código para un enrutador sería así:

```
// para establecer las distintas rutas, necesitamos instanciar
var router = express.Router()

//establecemos nuestra primera ruta, mediante get.
router.get('/', (req, res) => {
  res.json({ mensaje: '¡Bienvenido a nuestra API!' })
})

// nuestra ruta irá en http://localhost:8080/api
// es bueno que haya un prefijo, sobre todo por el tema de ver
app.use('/api', router)
```

CONFIGURA ENRUTADORES

- Crea un enrutador para el versionado de la API:
  - Ruta **GET /api** (simulará el versionado de la api):

```
{ mensaje: '¡Bienvenido a nuestra API!' }
```

- Crea un enrutador anidado para los endpoints de las cervezas
  - *GET /api/cervezas*
  - *POST /api/cervezas*
  - ...

SERVER.JS CON ENRUTADOR

```
var express = require('express') // llamamos a Express
var app = express()

var port = process.env.PORT || 8080 // establecemos nuestro pu

var router = require('./routes')

app.get('/', (req, res) => {
  res.json({ mensaje: '¡Hola Mundo!' })
})

app.use('/api', router)

// iniciamos nuestro servidor
app.listen(port, () => {
```

ENRUTADOR BASE PARA EL VERSIONADO

```
var router = require('express').Router()
var cervezasRouter = require('./cervezas')

router.get('/', (req, res) => {
  res.json({ mensaje: 'Bienvenido a nuestra api' })
})

router.use('/cervezas', cervezasRouter)

module.exports = router
```

ENRUTADOR CERVEZAS

```
var router = require('express').Router()

router.get('/', (req, res) => {
  res.json({ mensaje: 'Listado de cervezas' })
})

router.post('/', (req, res) => {
  res.json({ mensaje: 'Cerveza guardada' })
})

router.delete('/', (req, res) => {
  res.json({ mensaje: 'Cerveza borrada' })
})

module.exports = router
```

ENVIO DE PARÁMETROS

- Cuando el router recibe una petición, podemos observar que ejecuta una función de callback:

```
(req, res) => {}
```

  - El parámetro **req** representa la petición (request)
    - Aquí es donde se recibe el parámetro

TIPOS DE ENVIO DE PARÁMETROS

PARÁMETROS POR URL

- Vamos a mandar un parámetro *nombre* a nuestra api, de modo que nos de un saludo personalizado.

```
GET http://localhost:8080/pepito
```

```
app.get('/:nombre', (req, res) => {
  res.json({ mensaje: '¡Hola' + req.params.nombre })
})
```

- La variable podría acabar en ? (parámetro opcional)

- Si mandamos una url del tipo:

```
GET http://localhost:8080/api?nombre=pepito
```

- El parámetro se recoge mediante *req.query*:

```
app.get('/', (req, res) => {
  res.json({ mensaje: '¡Hola' + req.query.nombre })
})
```

PARÁMETROS POR POST

- ¡Necesitamos **middlewares**!
- **application/x-www-form-urlencoded**:
  - body-parser: extrae los datos del body y los convierte en json
- **multipart/form-data**
  - Busboy o Multer

EJEMPLO CON BODY-PARSER

- Instalación:

```
npm i -S body-parser@1.18.3
```

- body-parser actúa como **middleware**
- El código adicional será similar al siguiente:

```
var bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())

router.post('/', (req,res) => {
  res.json({mensaje: req.body.nombre})
})
```

RUTAS DE NUESTRA API

Ruta	verbo http	Descripción
/api/cervezas	GET	Obtenemos todas las cervezas
/api/cervezas/search?q=keyword	GET	Obtenemos cervezas por keyword
/api/cervezas/:id	GET	Obtenemos una cerveza
/api/cervezas	POST	Damos de alta una cerveza
/api/cervezas/:id	PUT	Actualizamos los datos de una cerveza
/api/cervezas/:id	DELETE	Borramos los datos de una cerveza

ENRUTADO DEL RECURSO CERVEZAS

- Intenta configurar una API básica para el recurso cervezas en base a las rutas anteriores
  - Muestra por consola el tipo de petición
  - Muestra por consola el parámetro de entrada

SOLUCIÓN ENRUTADO RECURSO CERVEZAS

- Fichero *app/routes/cervezas.js*:

```
var router = require('express').Router()
router.get('/search', (req, res) => {
  res.json({ message: `Vas a buscar una cerveza que contiene
}) // ¡¡¡¡antes que la ruta GET /:id!!!!
router.get('/', (req, res) => {
  res.json({ message: 'Estás conectado a la API. Recurso: ce
})
router.get('/:id', (req, res) => {
  res.json({ message: `Vas a obtener la cerveza con id ${req
})
router.post('/', (req, res) => {
  res.json({ message: 'Vas a añadir una cerveza' })
})
router.put('/:id', (req, res) => {
  res.json({ message: `Vas a actualizar la cerveza con id ${
```

ARQUITECTURA

SITUACIÓN ACTUAL

- Se han definido recursos
  - *Cervezas*, pero podría haber muchos más
  - Cada recurso se asocia a un enrutador
    - Por el momento *routes/cervezas*
  - A cada enrutador se asocian las rutas del recurso

ARQUITECTURA MVC

- Cada ruta se gestiona por un controlador
  - El controlador es responsable de:
    - Acceder a los datos para leer o guardar
      - Delega en un modelo
        - Utilizaremos un ODM
        - Mongoose para MongoDB
    - Enviar datos de vuelta
      - Al ser JSON, lo haremos directamente desde el controller
      - No necesitamos la capa Vista :-)

FAT MODEL, THIN CONTROLLER

- El controlador recoge la lógica de negocio.
  - En nuestro caso es muy sencillo:
    - Recoge parámetros
    - Llama al modelo (obtener datos, guardar)
    - Devuelve json
- El modelo puede ser complejo.
  - Por ej. creación de tokens o encriptación de passwords en el modelo de Usuario
  - Es un código que puede ser reutilizado entre controladores

ACCESO A BASE DE DATOS

- Para la persistencia de nuestros datos utilizaremos una base de datos
- Elegimos una noSQL: MongoDB
  - Es lo más habitual en arquitecturas MEAN
  - Así operamos con objetos json tanto en node como en bbdd (bson)
  - Y así tenemos otra consola de JavaScript :-)

INSTALACIÓN DE MONGODB

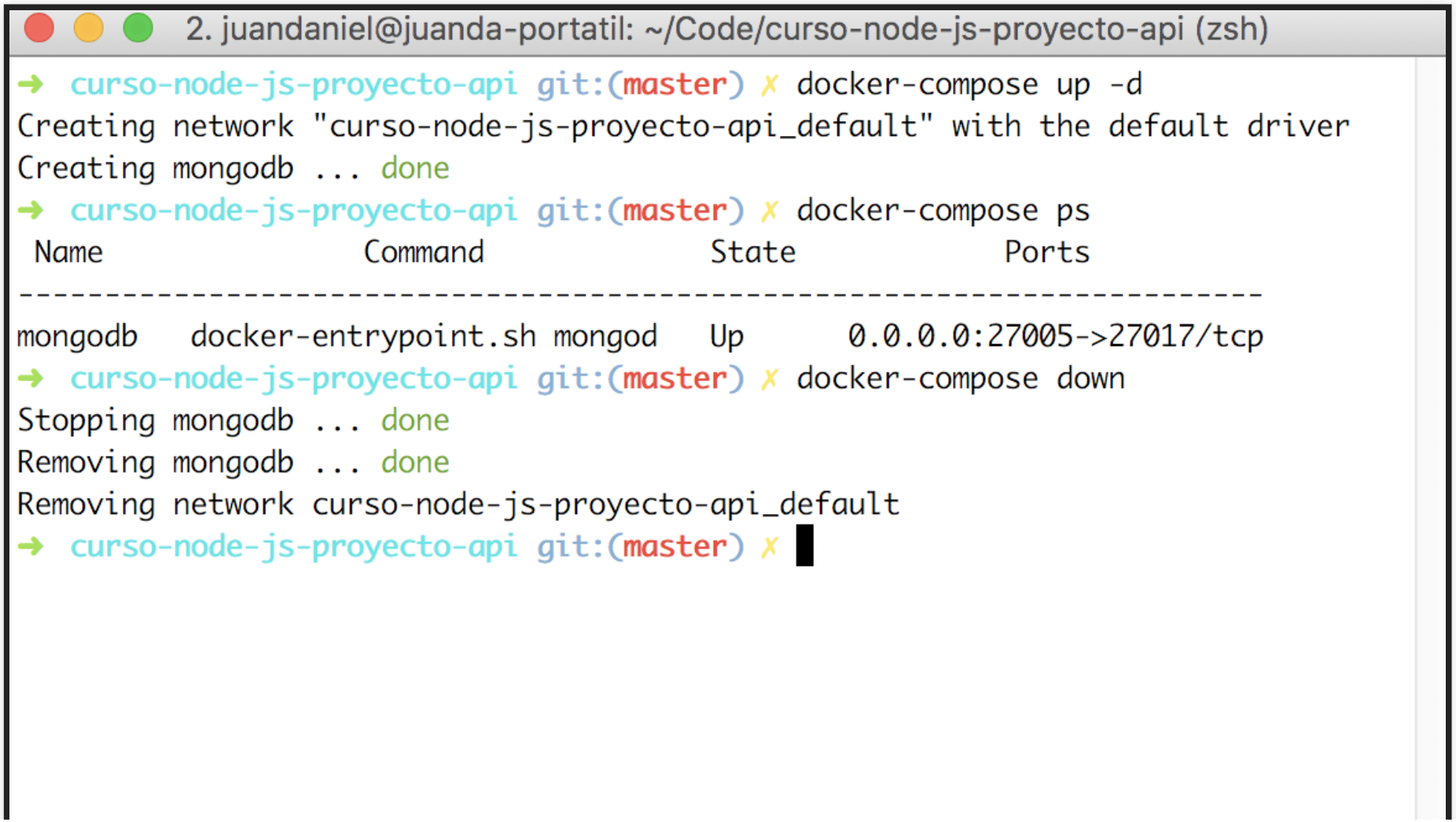
- Utilizaremos **contenedores docker**:
  - Eliminamos conflictos en la máquina base
  - Podemos cambiar de versiones con facilidad
  - Nuestro proyecto es más portable
- Docker ya está instalado
- Otra opción más tradicional sería usar repositorios o paquetes

FICHERO DE INSTALACIÓN MEDIANTE DOCKER

- Definiremos un fichero *docker-compose.yml*
- Para ver que imagen necesitamos podemos consultar en el docker hub

```
version: '3'
services:
  mongodb:
    hostname: mongodb
    container_name: mongodb
    image: mongo:4.0.1
    volumes:
      - ./mongodb-data:/data/db
    ports:
      - "27005:27017"
```

ARRANCAR Y PARAR MONGODB CON DOCKER-COMPOSE



MONGODB: CONSOLA

- Se ejecuta con el comando mongo
  - Como hemos instado MongoDB mediante docker, primero entraremos al contenedor:

```
docker-compose exec mongodb bash
mongo
```

    - O mediante *attach shell* de la extensión Docker de Visual Editor
  - Podemos ver:
    - Los ejecutables de MongoDB en el contenedor
    - El volumen mapeado...

MONGODB: APLICACIONES GRÁFICAS

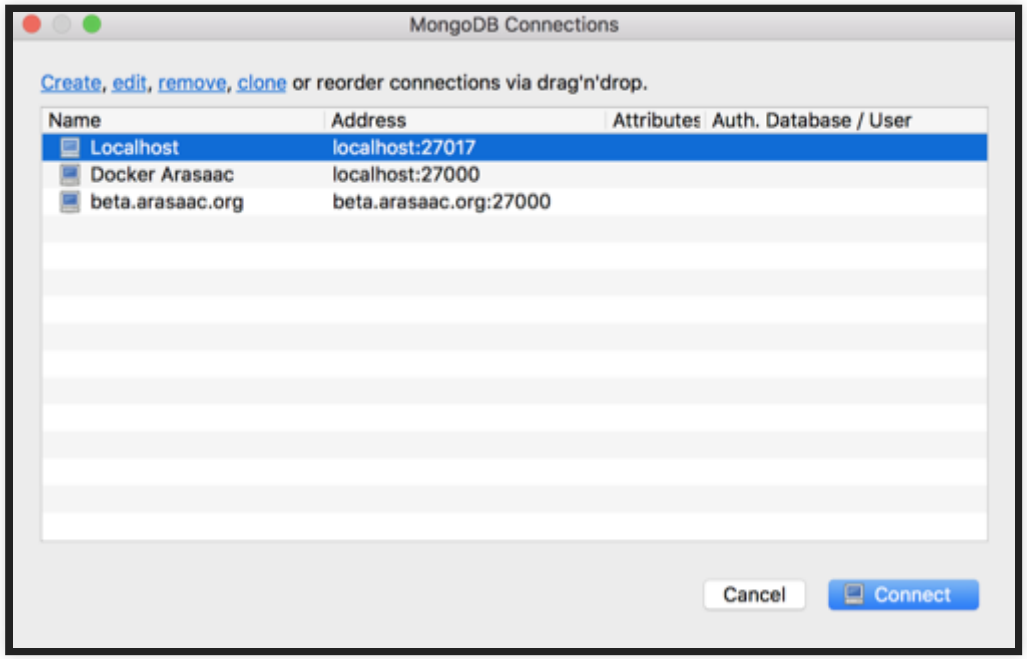
- mongo-express
  - Paquete de node
  - Instalación docker
- Robo3T
  - Antes llamado **Robomongo**
  - El más extendido, será el que utilicemos

ROBO3T: INSTALAR Y CONFIGURAR

- Descargamos el paquete de Robo3T(antes Robomongo)
- Instalamos y ejecutamos

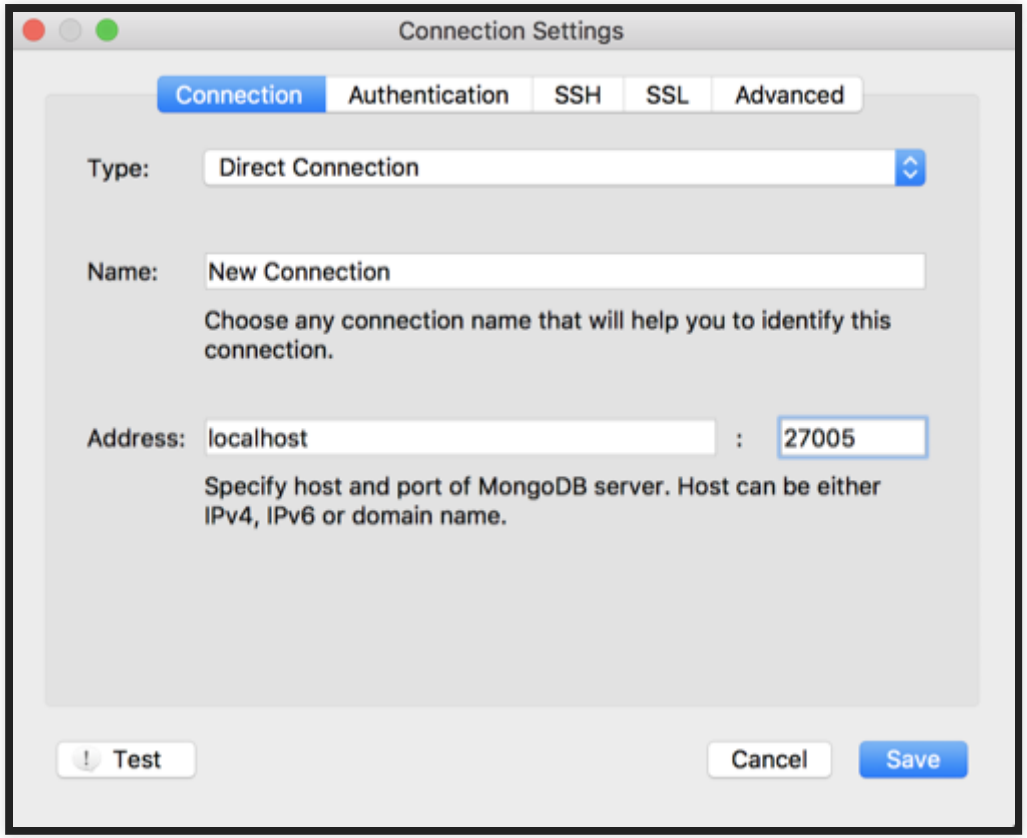
ROBO3T: CONEXIONES A MONGODB

- Robo3T guarda un listado de conexiones a MongoDB

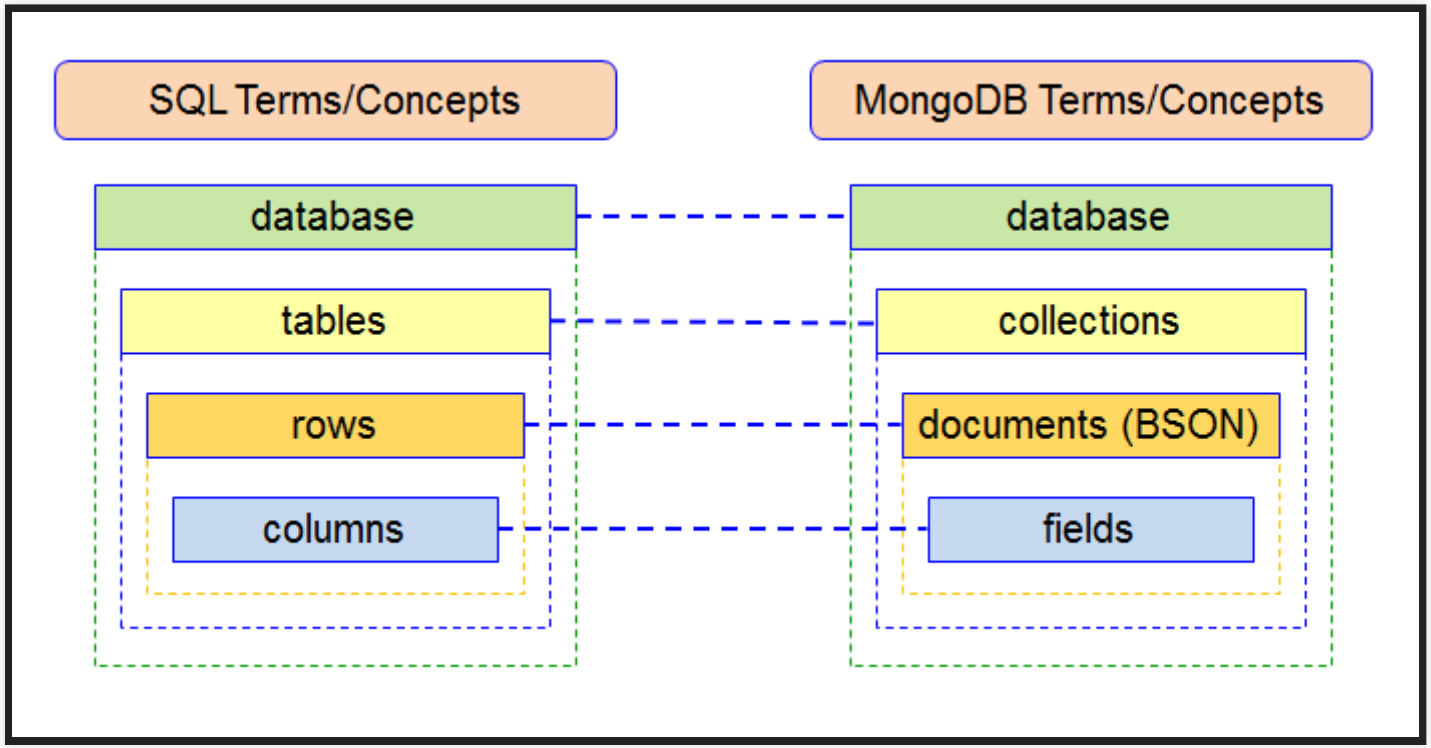


ROBO3T: CONFIGURAR CONEXIÓN A MONGODB

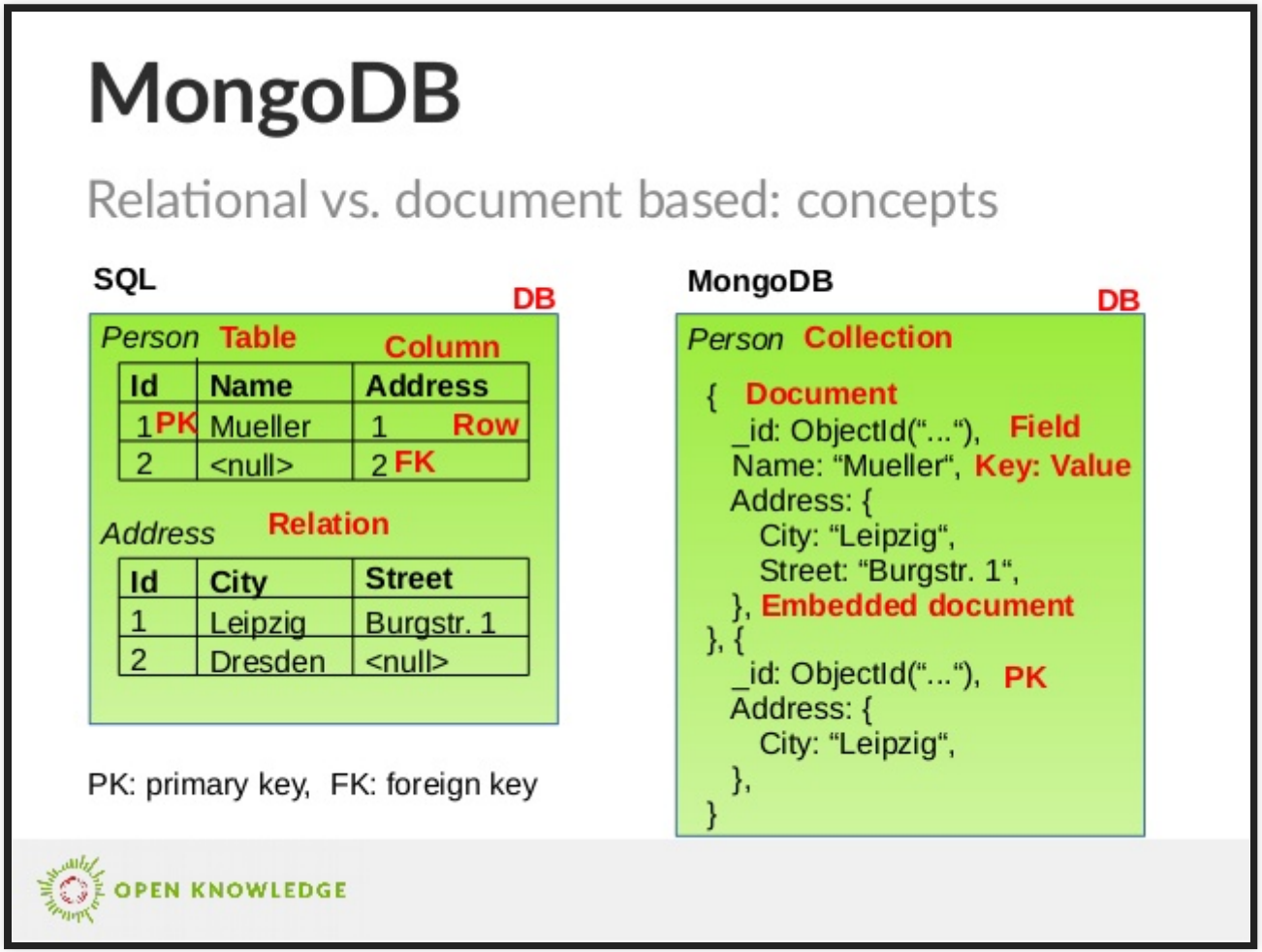
- Creamos una nueva conexión a localhost y al puerto que hemos mapeado en Docker (27005)


Connection Settings
Connection
Authentication
SSH
SSL
Advanced
Type:
Direct Connection
Name:
New Connection
Choose any connection name that will help you to identify this connection.
Address:
localhost
:
27005
Specify host and port of MongoDB server. Host can be either IPv4, IPv6 or domain name.
Test
Cancel
Save

CONCEPTOS EN NOSQL



SCHEMA EN NOSQL



INSERCIÓN DE DATOS

- Utilizaremos el fichero *cervezas.json*, de la carpeta data
- Importar nuestro cervezas.json a una base de datos

```
mongoimport --db web --collection cervezas --drop --file cerve
```

  - Otra opción es mediante Robomongo:

```
db.getCollection('cervezas').insertMany(array de objetos)
```

- Para hacer una búsqueda por varios campos de texto, tendremos que hacer un índice:

```
$ mongo # para entrar en la consola de mongo
use web; #seleccionamos la bbdd
db.cervezas.createIndex(
  {
    "nombre": "text",
    "descripción": "text"
  },
  {
    name: "CervezasIndex",
    default_language: "spanish"
  }
)
```

- Comprobamos que el índice esté bien creado

```
db.cervezas.getIndexes()
```

CONEXIÓN A MONGODB DESDE NODE

- Instalaremos mongoose como ODM (Object Document Mapper) en vez de trabajar con el driver nativo de MongoDB (se utiliza por debajo).

```
npm i -S mongoose@5.2.13
```

ABRIR CONEXIÓN CON MONGOOSE

- Mediante el método connect, siguiendo la documentación:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');

const Cat = mongoose.model('Cat', { name: String });

const kitty = new Cat({ name: 'Zildjian' });
kitty.save().then(() => console.log('meow'));
```

- Creamos el fichero *app/db.js* donde configuraremos nuestra conexión a base de datos mediante mongoose:

```
const mongoose = require('mongoose')

const MONGO_URL = process.env.MONGO_URL || 'mongodb://localhos
mongoose.connect(MONGO_URL, { useNewUrlParser: true })

mongoose.connection.on('connected', () => {
  console.log(`Conectado a la base de datos: ${MONGO_URL}`)
})

mongoose.connection.on('error', (err) => {
  console.log(`Error al conectar a la base de datos: ${err}`)
})

mongoose.connection.on('disconnected', () => {
  console.log('Desconectado de la base de datos')
```

- En nuestro fichero *app/server.js* incluimos el fichero de configuración de bbdd:

```
require('./db')
```

- Observa que no es necesario asignar el módulo a una constante
  - El módulo solo abre conexión con la bbdd
  - Registra eventos de mongodb
  - No exporta nada

- La conexión a bbdd se queda abierta durante todo el funcionamiento de la aplicación:
  - Las conexiones TCP son caras en tiempo y memoria
  - Se reutiliza

MODELOS

- Definimos un esquema para nuestra colección
- Creamos nuestro modelo a partir del esquema
- Fichero *app/models/Cervezas.js*):

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema

const cervezaSchema = new Schema({
  nombre: {
    type: String,
    required: true
  },
  descripción: String,
  graduación: String,
  envase: String,
  precio: String
})

const Cerveza = mongoose.model('Cerveza', cervezaSchema)
```

- Ahora podríamos crear documentos y guardarlos en la base de datos

```
const miCerveza = new Cerveza({ name: 'Ambar' })
miCerveza.save((err, miCerveza) => {
  if (err) return console.error(err)
  console.log(`Guardada en bbdd ${miCerveza.name}`)
})
```

GUARDAR DOCUMENTO

- Crea una cerveza nueva con todos los campos
- Comprueba desde Robo3T que en nuevo documento aparece en la colección Cervezas

SOLUCIÓN GUARDAR DOCUMENTO

```
require('./db')

// en fichero app/server.js después de conectar a bbdd:

const miCerveza = new Cerveza({
  nombre: 'Ambar',
  descripción: 'La cerveza de nuestra tierra',
  graduación: '4,8º',
  envase: 'Botella de 75cl',
  precio: '3€'
})
miCerveza.save((err, miCerveza) => {
  if (err) return console.error(err)
  console.log(`Guardada en bbdd ${miCerveza.nombre}`)
})
```

USO DE CONTROLADORES

- Desde nuestro fichero de rutas (*app/routes/cervezas.js*), se llama a un controlador, encargado de añadir, borrar o modificar cervezas usando el modelo Cerveza.
- **Endpoint -> Recurso -> Fichero de rutas -> Controlador -> Modelo**

- Creamos un directorio específico para los controladores (*app/controllers*)
- Un controlador específico para cada recurso, por ej. (*app/controllers/cervezasController.js*):
- Un método en el controlador por cada endpoint del recurso

REPASO DE NUESTRA API

Ruta	verbo http	Descripción
/api/cervezas	GET	Obtenemos todas las cervezas
/api/cervezas/search?q=keyword	GET	Obtenemos cervezas por keyword
/api/cervezas/:id	GET	Obtenemos una cerveza
/api/cervezas	POST	Damos de alta una cerveza
/api/cervezas/:id	PUT	Actualizamos los datos de una cerveza
/api/cervezas/:id	DELETE	Borramos los datos de una cerveza

CONFIGURACIÓN FINAL DEL ROUTER CERVEZAS

```
var router = require('express').Router()
var cervezasController = require ('../controllers/cervezasCont

router.get('/search', (req, res) => {
  cervezasController.search(req, res)
})
router.get('/', (req, res) => {
  cervezasController.list(req, res)
})
router.get('/:id', (req, res) => {
  cervezasController.show(req, res)
})
router.post('/', (req, res) => {
  cervezasController.create(req, res)
})
```

IMPLEMENTACIÓN DEL CONTROLADOR

WORKFLOW

- Utilizaremos enfoque BDD
  - Desarrollamos un test
  - Implementamos código
  - Comprobamos funcionamiento
- Los tests ya están hechos :-)

TEST DESDE EL NAVEGADOR O MEDIANTE POSTMAN

- Comprobamos que se genera el listado de cervezas
- Comprobamos que se busca por keyword:

```
http://localhost:8080/api/cervezas/search?q=regaliz
```

...

TEST DE LA API

- Utilizaremos Mocha como test framework
- supertest para hacer las peticiones http.
- Chai como librería de aserciones

```
npm i -D mocha supertest chai
```

- Tenemos nuestros test en el fichero *test/cerveza.test.js*

CONFIGURAMOS NUESTRO APP PARA LOS TESTS

```
// iniciamos nuestro servidor
// para tests, porque supertest hace el listen por su cuenta
if (!module.parent) {
  app.listen(port, () => console.log(`API escuchando en el pue
}

// exportamos la app para hacer tests
module.exports = app
```

CONFIGURACIÓN DEL CONTROLADOR CERVEZAS

- Debemos definir los métodos siguientes:
  - search
  - list
  - show
  - create
  - update
  - remove

LISTAR CERVEZAS

```
const Cerveza = require('../models/Cerveza')

const list = (req, res) => {
  Cerveza.find((err, cervezas) => {
    if (err) {
      return res.status(500).json({
        message: 'Error obteniendo la cerveza'
      })
    }
    return res.json(cervezas)
  })
}

module.exports = {
  list
```

LISTAR CERVEZAS POR PALABRA CLAVE

```
const Cerveza = require('../models/Cerveza')
const search = (req, res) => {
  const q = req.query.q
  Cerveza.find({ $text: { $search: q } }, (err, cervezas) => {
    if (err) {
      return res.status(500).json({
        message: 'Error en la búsqueda'
      })
    }
    if (!cervezas.length) {
      return res.status(404).json({
        message: 'No hemos encontrado cervezas que cumplan esa
      })
    } else {
      return res.json(cervezas)
```

MOSTRAR UNA CERVEZA

```
const Cerveza = require('../models/Cervezas')
const { ObjectId } = require('mongodb')
const show = (req, res) => {
  const id = req.params.id
  Cerveza.findOne({ _id: id }, (err, cerveza) => {
    if (!ObjectId.isValid(id)) {
      return res.status(404).send()
    }
    if (err) {
      return res.status(500).json({
        message: 'Se ha producido un error al obtener la cerve
      })
    }
    if (!cerveza) {
      return res.status(404).json({
```

CREAR UNA CERVEZA

```
const Cerveza = require('../models/Cervezas')
const create = (req, res) => {
  const cerveza = new Cerveza(req.body)
  cerveza.save((err, cerveza) => {
    if (err) {
      return res.status(400).json({
        message: 'Error al guardar la cerveza',
        error: err
      })
    }
    return res.status(201).json(cerveza)
  })
}
module.exports = {
  search,
```

ACTUALIZAR CERVEZA

```
const update = (req, res) => {
  const id = req.params.id
  Cerveza.findOne({ _id: id }, (err, cerveza) => {
    if (!ObjectId.isValid(id)) {
      return res.status(404).send()
    }
    if (err) {
      return res.status(500).json({
        message: 'Se ha producido un error al guardar la cerve
        error: err
      })
    }
    if (!ObjectId.isValid(id)) {
      return res.status(404).send()
    }
```

BORRAR CERVEZA

```
const Cerveza = require('../models/Cervezas')
const { ObjectId } = require('mongodb')
const remove = (req, res) => {
  const id = req.params.id

  Cerveza.findOneAndDelete({ _id: id }, (err, cerveza) => {
    if (!ObjectId.isValid(id)) {
      return res.status(404).send()
    }
    if (err) {
      return res.json(500, {
        message: 'No hemos encontrado la cerveza'
      })
    }
    if (!cerveza) {
```

ANÁLISIS DE CÓDIGO

- Por último podríamos utilizar un paquete como **istanb** código y ver si nuestras pruebas recorren todas las ins ramas del código:

```
npm i -D istanbul
./node_modules/.bin/istanbul cover -x "**/tests/**"  ./node_mo
```

- Estos datos son facilmente exportables a algún servicio que nos de una estadística de la cobertura de nuestros tests o que haga un seguimiento de los mismos entre las distintas versiones de nuestro código.
- Por último también se podría integrar con un sistema de integración continua tipo Travis.

USO DE MIDDLEWARES CORS Y MORGAN

- Normalmente utilizaremos middlewares que ya están hechos, por ejemplo Morgan para logs y cors para Cors.
- Los instalamos:

```
npm i -S cors morgan
```

- Los insertamos en nuestra API (el orden puede ser importante):

```
const express = require('express') // llamamos a Express
const app = express()
const router = require('./routes')

const cors = require('cors')
const morgan = require('morgan')
app.use(morgan('combined'))
app.use(cors())

require('./db')

const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: true }))
app.use(bodyParser.json())
```