DESARROLLO WEB FULL STACK

Construcción de API's

Interfaces cliente-servidor

DWFS COR

¿Qué es una API?

- Es una forma de describir la forma en que los programas o los sitios webs intercambian datos.
- El formato de intercambio de datos normalmente es **JSON** o **XML**.

API REST

- REST viene de, **RE**presentational **S**tate **T**ransfer
- Es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP.
- REST se compone de una lista de reglas que se deben cumplir en el diseño de la arquitectura de una API.
- Hablaremos de servicios web restful si cumplen la arquitectura REST.
- Restful = adjetivo, Rest = Nombre

Recurso

Recurso es la **URI** con la cual se efectúa la petición HTTP.

Convenciones de nombrado

Usar sustantivos para representar recursos

La **URI** en RESTful debe referirse a un recurso que es una **cosa** (sustantivo) en lugar de referirse a una acción (verbo) porque los sustantivos tienen propiedades que los verbos no tienen, de forma similar a los recursos tienen atributos. Algunos ejemplos de un recurso son:

- Usuarios de un sistema
- Cuentas de usuarios
- Dispositivos de red

Y sus recursos pueden ser descritos como a continuación:

http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados
http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados/{id}
http://api.ejemplo.com/administracion-usuarios/usuarios/
http://api.ejemplo.com/administracion-usuarios/usuarios/{id}

Arquetipos de recursos

Para mayor claridad, dividamos los arquetipos de recursos en cuatro categorías:

- Document
- Collection
- Store
- Controller

Document

Un recurso de documento es un concepto singular que es similar a una instancia de objeto o registro de base de datos. En REST, puede verlo como un único recurso dentro de la colección de recursos. La representación del estado de un documento generalmente incluye ambos campos con valores y enlaces a otros recursos relacionados.

Utilice el nombre "singular" para denotar el arquetipo de recursos del documento.

Ej.:

http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados/{id}
http://api.ejemplo.com/administracion-usuarios/usuarios/{id}
http://api.ejemplo.com/administracion-usuarios/usuarios/admin

Collection

Un recurso de colección es un directorio de recursos **administrado por el servidor**. Depende de la colección elegir crear un nuevo recurso, o no. Un recurso de colección elige lo que quiere contener y también decide los URI de cada recurso contenido.

Utilice el nombre "plural" para denotar el arquetipo de recurso de colección.

Ej.:

http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados
http://api.ejemplo.com/administracion-usuarios/usuarios
http://api.ejemplo.com/administracion-usuarios/usuarios/{id}/accounts

Store

Un **store** es un repositorio de recursos **administrado por el cliente**. Un recurso de store permite que un cliente API ingrese recursos, los retire y decida cuándo eliminarlos. Un store nunca genera nuevos URI.

Utilice el nombre "plural" para denotar el arquetipo de recursos de store.

Ej.:

http://api.ejemplo.com/administracion-carritos/usuarios/{id}/carritos
http://api.ejemplo.com/administracion-canciones/usuarios/{id}/listas

Controller

Un recurso controlador modela un concepto de procedimiento. Los recursos del controlador son como funciones ejecutables, con parámetros y valores de retorno; entradas y salidas.

Use "verbo" para denotar el arquetipo del controlador.

Ej.:

http://api.ejemplo.com/administracion-carritos/usuarios/{id}/carrito/comprar
http://api.ejemplo.com/administracion-canciones/usuarios/{id}/lista/reproducir

Consistencia es la clave

- **Utilice la barra diagonal (/) para indicar relaciones jerárquicas**

  http://api.ejemplo.com/administracion-dispositivos
  http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados
  http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados/{id}

- **No utilice la barra diagonal final (/) en los URI**

  http://api.ejemplo.com/administracion-dispositivos/ *INCORRECTO*
  http://api.ejemplo.com/administracion-dispositivos *CORRECTO*

- **Use guiones (-) para mejorar la legibilidad de los URI**

  http://api.ejemplo.com/administracion-dispositivos

Consistencia es la clave

- **No use guiones bajos (_)**

  http://api.ejemplo.com/administracion_dispositivos *INCORRECTO*
  http://api.ejemplo.com/administracion-dispositivos *CORRECTO*

- **Use letras minúsculas en URI**

  http://api.ejemplo.com/administracion-dispositivos

- **No use extensiones de archivo**

  http://api.ejemplo.com/administracion-carritos/usuarios/{id}/carritos.xml *INCORRECTO*
  http://api.ejemplo.com/administracion-carritos/usuarios/{id}/carritos *CORRECTO*

Consistencia es la clave

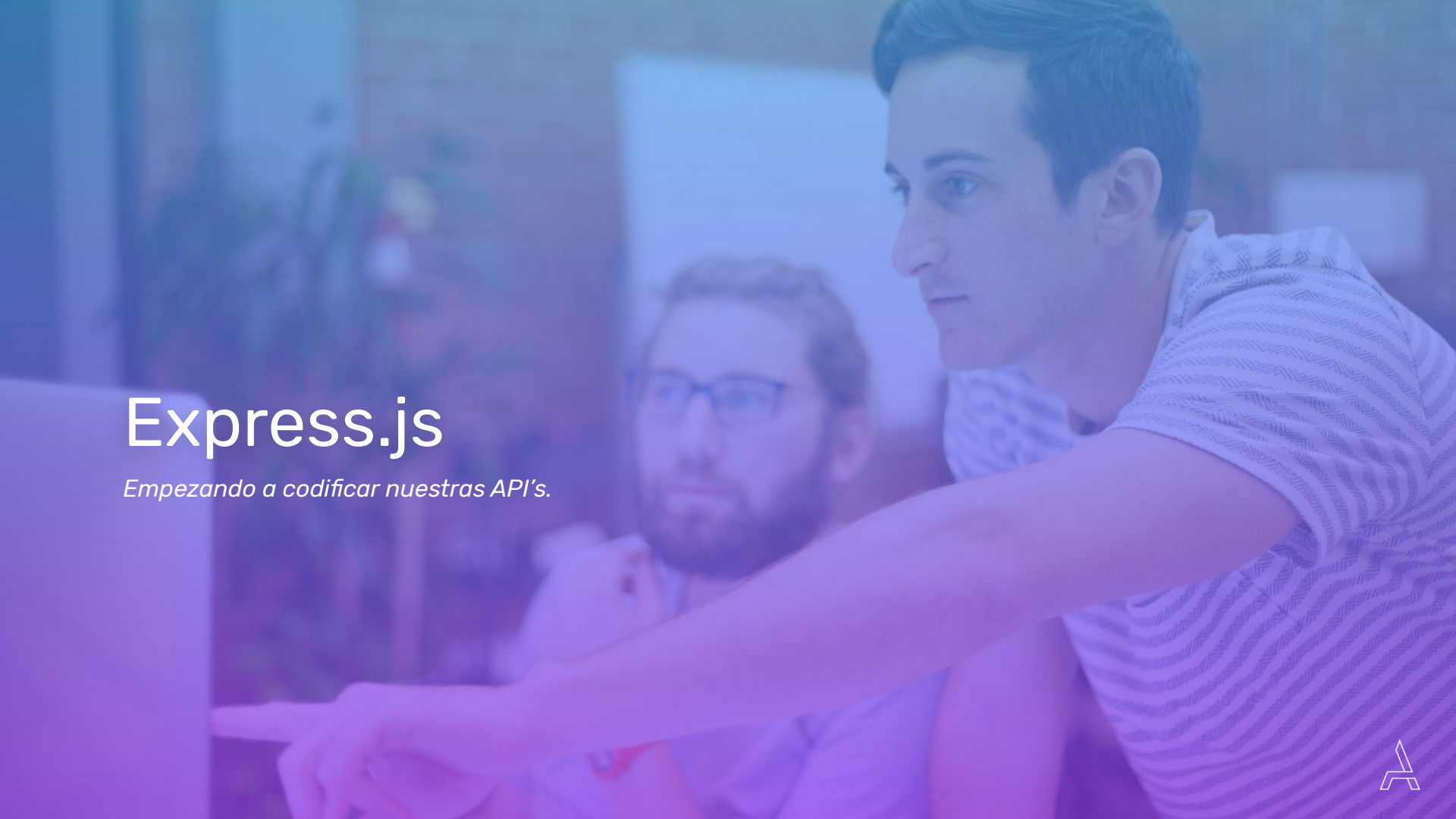
- **Nunca use nombres de funciones CRUD en URI**

  INCORRECTO
  http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados/crear
  http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados/borrar
  http://api.ejemplo.com/administracion-dispositivos/dispositivos-administrados/modificar

- **Use query params para filtrar la colección de URI**

  http://api.ejemplo.com/administracion-canciones/usuarios/{id}/lista?region=AR

Práctica:
Diseñar los recursos de API para una hipotética API de lista de tareas. Tener en cuenta nuestro diseño de base de datos.



Express.js

Empezando a codificar nuestras API's.

¿Qué es Express.js?

Express.js, o simplemente Express, es un framework de aplicación web para Node.js, lanzado como software gratuito y de código abierto bajo la Licencia MIT.

Está diseñado para crear aplicaciones web y API. Se le ha llamado el framework de servidor estándar de facto para Node.js.

```
npm install express
```

https://www.npmjs.com/package/express

Primera aplicación con express

Express nos crea un servidor y cuando se use una URI que coincida con la ruta declarada como primer parámetro, ejecuta la función de callback del segundo parámetro.

```
const express = require('express');
const app = express();

app.get('/tareas', function (req, res) {
  res.send('¡Te mando una tarea!')
});

app.listen(3000, function () {
  console.log('La API está viva!!!! >:D');
});
```

Objetos **req** y **res**

req: representa la **solicitud** HTTP y tiene propiedades para todo lo relacionado a la solicitud, parámetros, cuerpo, encabezados HTTP, etc.

res: representa la respuesta HTTP que envía una aplicación Express cuando recibe una solicitud HTTP.

Nota: por convención los objetos se deben llamar así en el código de aplicación.

Ruteo

El ruteo hace referencia a la determinación de cómo responde una aplicación a una solicitud de cliente en un determinado punto final, que es un URI (o una vía de acceso) y un método de solicitud HTTP específico (GET, POST, etc.)

Estructura:

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

- **app:** Instancia de express.
- **METHOD:** El método de solicitud HTTP.
- **PATH:** Una vía de acceso al servidor.
- **HANDLER:** Función que se ejecuta cuando se correlaciona la ruta.

Ruteo

Dos rutas con método GET.

```
const express = require('express');
const app = express();

app.get('/ejemplo', function (req, res) {
  res.send('Estoy sobre la ruta ejemplo.')
});

app.get('/otra-ruta/ejemplo', function (req, res) {
  res.send('Otra ruta de ejemplo.')
});

app.listen(3000);
```

Node modules

Porción de código que se encuentra en un archivo diferente y puede ser incluída en cualquier parte de nuestro programa.

Son necesarios para la separación de responsabilidades dentro del software.

Palabras clave:

- **module.exports: Disponibiliza** una porción de código para ser incluída en cualquier otro archivo de nuestro programa.
- **require():** con esta función se puede **“traer”** cualquier porción de código **exportada** a un archivo nuevo donde queramos usarlo.

Modulo

Creo un módulo y exporto una función.

Nombre de archivo:
suma.js

```
var suma = function(a, b) {
     return a + b;
}

module.exports = suma;
```

Uso el módulo anterior

Importo/requiero/incluyo la funcionalidad que desarrollé en mi módulo.

```
var suma = require('./suma.js');

console.log(suma(1, 3));
```

Express router

Nos permite crear un archivo separado de rutas para hacer separación de responsabilidades.

Router

Dos rutas con método GET.

Nombre de módulo: **rutas.js**

```
const express = require('express');
const router = express.Router();

router.get('/ruta-1', function (req, res) {
  res.send('Estoy sobre la ruta ejemplo 1.')
});

router.get('/ruta-2', function (req, res) {
  res.send('Estoy sobre la ruta ejemplo 2.')
});

module.exports = router;
```

Router

Uso el router.

```
const express = require('express');
const app = express();
const rutas = require('./rutas.js');

app.use('/', rutas);

app.listen(3000);
```

Parámetros de ruta

Nos permiten definir en nuestras rutas parámetros que son variables, de esta forma podemos definir rutas dinámicas que recuperan otra lógica de acuerdo al parámetro pasado.

Los mismos se encuentran en el objeto **req** dentro del atributo **params**.

Ejemplo:

```
app.get('/ruta/:parametro', function (req, res) {
  res.send('El parámetro enviado fue: ' + req.params.parametro);
});
```

Parámetros de consulta

Nos permiten definir en nuestras rutas configuraciones que viajan con las rutas, una ruta puede contener todas las configuraciones que queramos, solo hay que tener en cuenta que estas configuraciones pueden o no venir, es decir, no son obligatorias.

Los mismos se encuentran en el objeto **req** dentro del atributo **query**.

Ejemplo:

```
app.get('/ruta-1, function (req, res) {
  res.send('La configuración enviada fue: ' + req.query.config);
});
```

Middlewares

Las funciones de middleware son funciones que se ejecutan en el medio del ciclo de solicitudes/respuestas manejadas con express.

Tienen **acceso** al **objeto de solicitud (req)**, al **objeto de respuesta (res)** y **a la siguiente función de middleware**. La siguiente función de middleware se denota con una variable denominada **next**.

Pueden realizar las siguientes tareas:

- Ejecutar cualquier código.
- Realizar cambios en la solicitud y los objetos de respuesta.
- Finalizar el ciclo de solicitud/respuestas.
- Invocar la siguiente función de middleware en la pila.

Middleware de aplicación

Se ejecutan a nivel de aplicación, antes de llegar a cualquier ruta en este caso.

```
const express = require('express');
const app = express();

app.use(function (req, res, next) {
  console.log('Me ejecuto antes de cada request');
  next();
});

app.get('/', function (req, res) {
  res.send('¡Hola mundo!')
});

app.listen(3000);
```

Middleware de ruta

Se ejecutan a nivel de ruta, es decir, solo para la ruta en la que son declarados, son funciones comunes y corrientes.

```
const express = require('express');
const app = express();


app.get('/',
  function (req, res, next) {
    console.log('Función de middleware de ruta');
    next();
  },
  function (req, res) {
    res.send('¡Hola mundo!')
  }
);

app.listen(3000);
```

Middleware de error

Se ejecutan a nivel de aplicación, se deben poner al final de todas nuestras rutas para que pueda capturar los errores que pasen en cualquiera de estas.

```
const express = require('express');
const app = express();


app.get('/', function (req, res) {
    res.send('¡Hola mundo!');
});

app.use(function (err, req, res, next) {
    res.status(500).send('Ocurrió un error.');
});

app.listen(3000);
```

Comunicación entre middlewares

La forma de pasar datos entre cadenas de middlewares es a través del objeto de **solicitud (req)** en la propiedad **locals**: **req.locals**

```
app.get('/ruta/:parametro',
  function (req, res, next) {
    res.locals.nombre = 'Pepito rojo';
    next();
  },
  function (req, res) {
    res.send(
      'El parámetro enviado fue: ' + req.params.parametro +
      'El nombre que viene del middleware es ' + res.locals.nombre
    );
  }
);
```