Facultad de Ciencias de la Administración

TECNICATURA UNIVERSITARIA EN DESARROLLO WEB

Creación de una API Rest

Semana N.º 3 – API Rest

Tecnicatura Universitaria en Desarrollo Web

Facultad de Ciencias de la Administración - Universidad Nacional de Entre Ríos

Objetivos de la clase

- **Objetivos**
  - Conectarse con un servidor de base de datos.
  - Utilizar comandos SQL apropiados para enviar y recuperar datos.
  - Diseñar y estructurar una aplicación Express.js modular, siguiendo buenas prácticas.
- **Temas a desarrollar:**
  - Repaso de conceptos de Bases de Datos: Lenguaje SQL. Creación de Tablas, Vistas y Procedimientos almacenados.
  - Conexión a una Base de Datos. Operaciones de selección, inserción, actualización y eliminación de datos.
  - Profundizando en Express.js: Rutas, Controladores y Middlewares.
  - Pruebas con Bruno/Postman. Documentación con Swagger.
  - Buenas prácticas en el diseño de una API Rest. Seguridad y manejo de errores

API Rest

- En **IDW** utilizamos **`fetch`** para recuperar datos desde una API externa. Llega el momento de saber cómo esas APIs son creadas.
- **REST** es el acrónimo de **RE**presentational **S**tate **T**ransfer. Define un estilo arquitectónico para diseñar sistemas distribuidos.
- El término fue introducido en la tesis doctoral de Roy Fielding en el año 2000.
- No es exactamente una arquitectura de software sino un ***conjunto de restricciones***:
  - Uniformidad.
  - Cliente – Servidor.
  - Sin estado.
  - Puede ser almacenado en una cache (cacheable).
  - Arquitectura en capas.
  - Código a demanda (opcional).

- Los términos **API REST, RESTful Web Service** y **REST Web Service** si bien no se refieren exactamente a lo mismo, son utilizados como sinónimos.

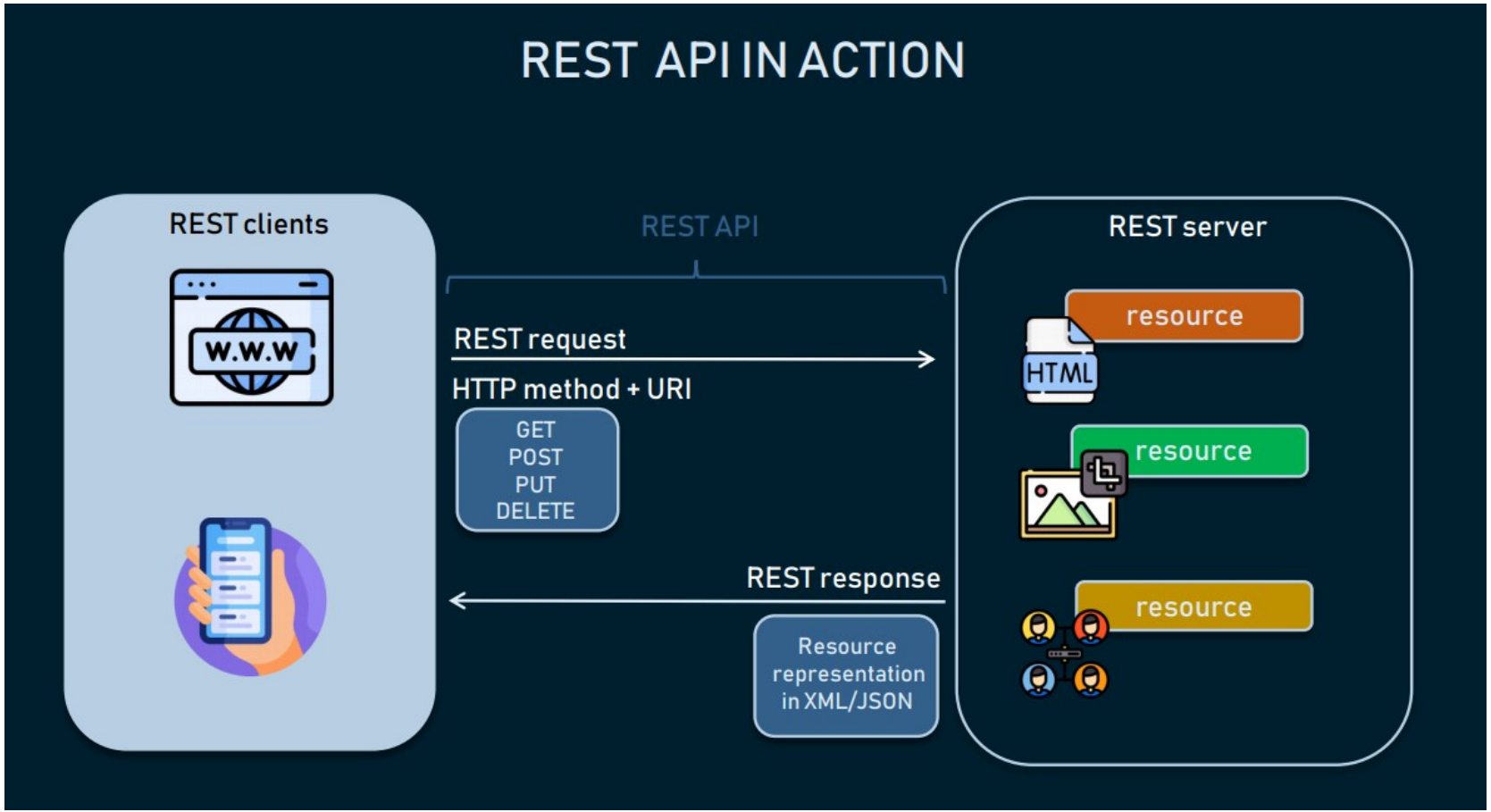
Recurso y representación

- Las abstracciones de información en **REST** se conocen con el nombre de ***Resource*** o ***Recurso***.
- Cada **recurso** debe tener un nombre que permita identificarlo.
- En **REST** el ***estado de un recurso*** se refleja en la ***representación*** del mismo, y las transiciones de estado se hacen mediante las interacciones cliente–servidor.
- El formato de una **representación** se llama ***Media Type*** y determina como la representación debe ser procesada. El formato más utilizado es **JSON**.
- En el contexto de **REST**, generalmente escuchamos el término ***endpoint***.
- Un ***endpoint*** hace referencia a una **URL** específica que representa un **recurso** o una **colección de recursos**.
  - Los ***endpoints*** son ubicaciones específicas a los que se puede enviar una solicitud **HTTP** para interactuar con recursos.

API REST – Resource Methods

- Fielding no estableció una recomendación respecto de los métodos pero enfatizó que la API debe tener nombres uniformes.
- Los métodos de recurso son por lo general asociados con los métodos **HTTP**: **GET, POST, PUT, DELETE:**
  - **GET**: Recupera la información. Puede ser una colección o una única entidad.
  - **POST**: Solicita que el recurso cree una nueva entidad.
  - **PUT**: actualiza una entidad.
  - **DELETE**: remueve o elimina el elemento.

Métodos HTTP y API REST



Restricciones Arquitectónicas

- **REST** es un estilo arquitectónico que sigue principios para garantizar interoperabilidad, escalabilidad y flexibilidad en sistemas distribuidos.
- Las principales restricciones son:
  - **Cliente-Servidor:** permite la separación de responsabilidades, lo que facilita la escalabilidad y el desarrollo independiente de ambos.
  - **Sin Estado:** Cada solicitud del cliente debe suministrar toda la información suficiente para que el servidor pueda procesarla. El servidor no debe almacenar el estado de solicitudes previas. Esto permite una mayor escalabilidad, ya que cada solicitud es independiente de las demás.
  - **Cacheable:** Las respuestas del servidor deben indicar si los datos pueden ser almacenados en caché por el cliente. Esto mejora el rendimiento y la eficiencia al reducir la carga de trabajo en el servidor y permitir que los clientes reutilicen los datos almacenados localmente.

Restricciones Arquitectónicas (2)

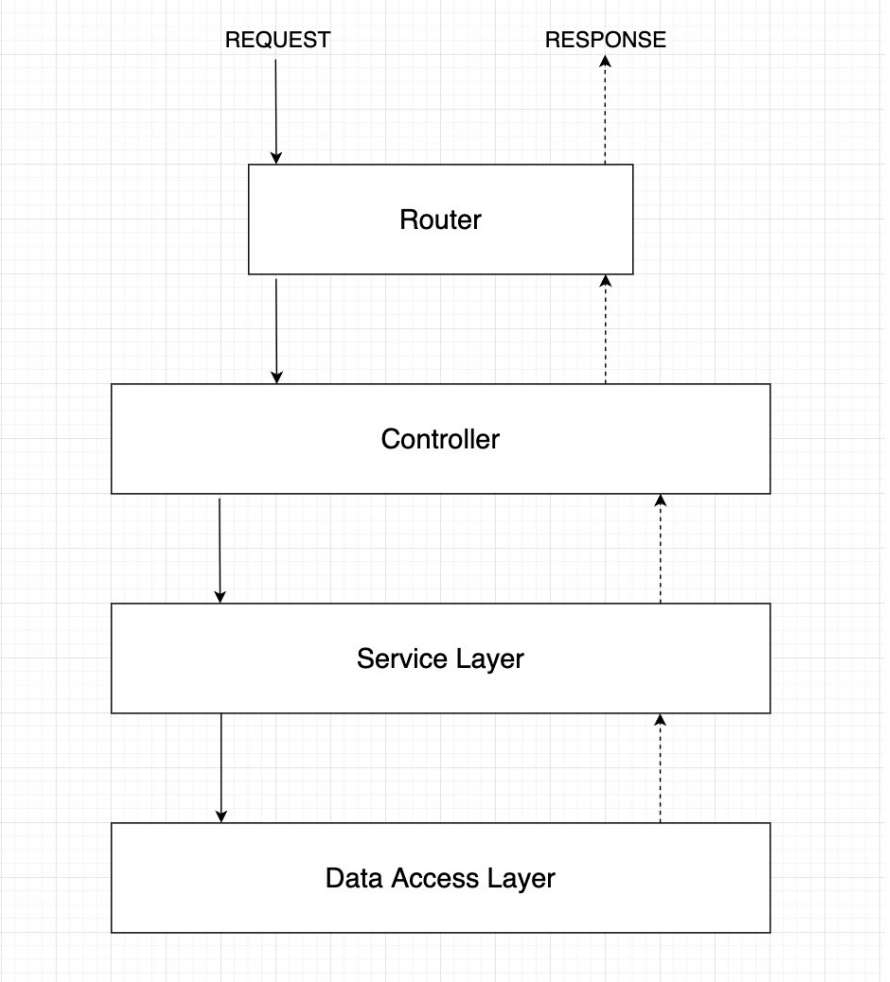
- **Interfaz Uniforme:** para que los componentes interactúen de forma coherente la interfaz debe ser uniforme. Esto incluye:
  - Identificación de recursos: Cada recurso es identificado mediante una URL única.
  - Manipulación de recursos a través de representaciones: Los clientes interactúan con los recursos mediante sus representaciones (JSON o XML).
  - Mensajes autodescriptivos: Cada solicitud y respuesta debe contener suficiente información para que sea comprensible por sí sola.
  - Hipermedios como el motor del estado de la aplicación (HATEOAS): respuestas incluyen enlaces a otros recursos o acciones que el cliente puede realizar, permitiendo la navegación dinámica.

```
{
  "idFactura": 123456,
  "estado": "Impaga",
  "total": 150000.00,
  "_links": {
    "self": {
      "href": "/api/facturas/123456"
    },
    "pagar": {
      "href": "/api/pagos/123456",
      "method": "POST"
    },
    "update": {
      "href": "/api/facturas/123456",
      "method": "PUT"
    },
    "items": {
      "href": "/api/facturas/123/items"
    }
  }
}
```

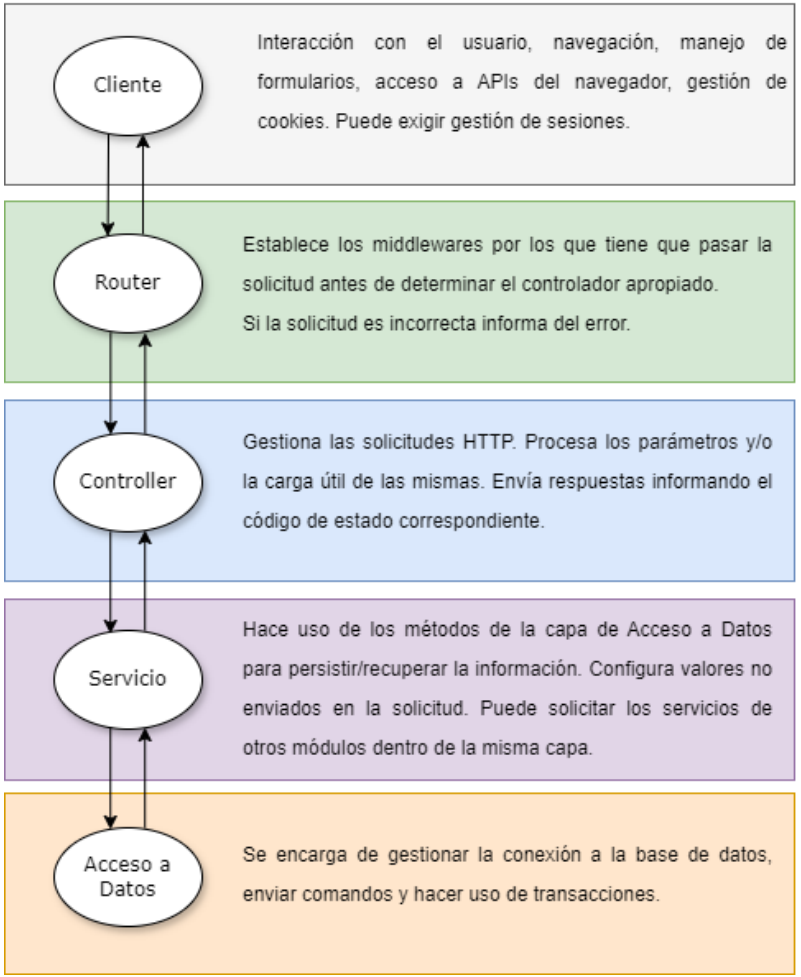
- **Sistema en Capas:** el cliente no necesita saber si está conectado directamente al servidor o a una capa intermedia (proxy o un balanceador de carga). Esto mejora la seguridad, escalabilidad y administración del sistema.
- **Código Bajo Demanda (Opcional):** Esta restricción permite al servidor proporcionar código ejecutable al cliente, por ejemplo, en forma de scripts **JavaScript**, lo que amplía las capacidades del cliente. Sin embargo, esta es una restricción opcional.

Mejores Prácticas - Arquitectura

- Definir una arquitectura con responsabilidades bien claras permite reutilizar código y responder más rápido a cambios de requerimientos.
- Un **router** de **Express** que pasa las solicitudes al **controlador** correspondiente.
- En el **Controlador** gestionaremos las solicitudes **HTTP** y entregaremos las respuestas correspondientes a cada **endpoint**.
- La ***lógica de negocio*** estará en la ***capa de servicio*** que exporta ciertos servicios (métodos) que utiliza el **controlador.**
- La tercera capa es la de **acceso a datos** donde trabajaremos con nuestra **base de datos**. En nuestro caso, utilizaremos una **base de datos relacional**: **MySQL.**
  - Cabe aclarar que podría tratarse de cualquier motor de base de datos **SQL, NoSQL** o incluso un archivos.



Mejores Prácticas – Arquitectura | Responsabilidades



Middleware

- En **Express** un ***middleware*** es un mecanismo para encapsular una funcionalidad que opera sobre una solicitud HTTP a nuestra aplicación.

- En la práctica, un **middleware** es simplemente una función que toma tres argumentos: un objeto **`request`** (solicitud), un objeto **`response`** (respuesta) y una función **`next()`**. (También se pueden usar cuatro argumentos, para el manejo de errores).

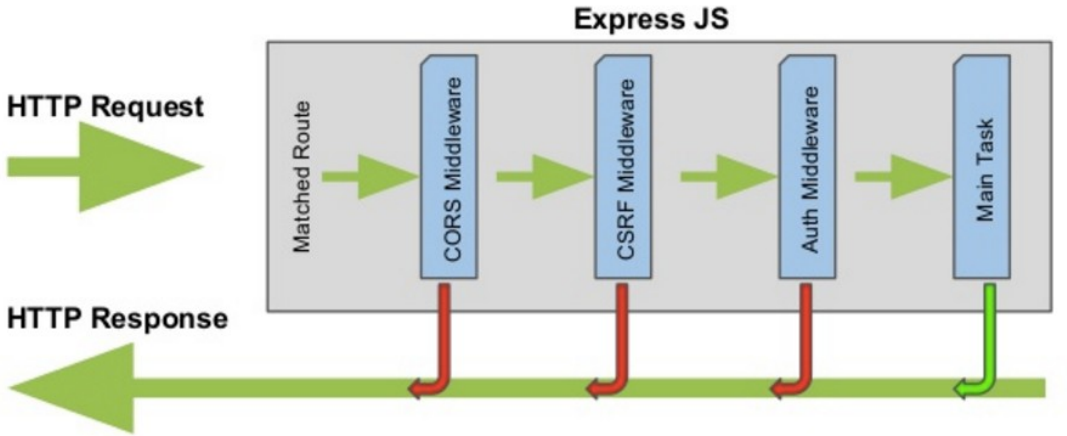
- Un **middleware** se ejecuta en lo que se conoce como pipeline. Podemos imaginarnos una tubería que transporta agua. El agua se bombea por un extremo y luego hay medidores y válvulas antes que el agua llegue a su destino.

- Siguiendo con la analogía, el orden importa. Si colocamos un medidor antes de una válvula, tiene un efecto diferente que si coloca el medidor después de la válvula.

- De manera similar, si tiene una válvula que inyecta algo en el agua, todo lo que esté “aguas abajo” de esa válvula contendrá el “ingrediente agregado”. En una aplicación **Express**, insertamos el **middleware** en el pipeline llamando a **`app.use`**.

Middleware - Principios

- A la hora de programar nuestros propios **middleware** tenemos que tener en consideración los siguientes principios:
  - Cada uno de las funciones **middleware** debe hacer una llamada a **`next()`** para que el pipeline no se termine. Si se omite la llamada a **`next()`** entonces es el final de la tubería.
  - Si no llamamos a **`next()`** deberíamos hacer un **`res.render()`** o **`res.send()`** para enviar algo al cliente.
  - Caso contrario no retornaremos nada y el cliente dará tiempo de espera agotado.
  - Si hacemos un **`next()`** no debemos hacer ningún **`res.render()`** o **`res.send()`**. Caso contrario las siguientes respuestas serán ignoradas por el cliente.



Middlewares de uso común

- **Seguridad:**
  - `basicauth-middleware, csurf, cookie-parser, express-session.`
- **Archivos:**
  - `busboy, multiparty, formidable, multer.`
- **Performance:**
  - `compression, response-time.`
- **Rutas/Recursos:**
  - `static, serve-favicon, serve-index, vhost.`
- **Debug/Dev:**
  - `morgan, method-override, errorhandler.`

Middlewares de uso común (2)

Middleware	Función principal	Ejemplo de uso típico
basicauth-middleware	Autenticación HTTP básica (usuario/contraseña)	Proteger un endpoint con `Authorization: Basic`
cookie-parser	Parsear cookies y dejarlas accesibles en `req.cookies`	`app.use(cookieParser('miSecreto'))`
express-session	Manejar sesiones de usuario en servidor (con cookies)	`app.use(session({ secret: 'clave' }))`
csurf	Protección contra ataques CSRF usando tokens	`app.use(csurf({ cookie: true }))`
multer	Middleware moderno y flexible para uploads de archivos	`app.post('/upload', upload.single('foto'))`
compression	Comprimir respuestas HTTP con gzip/deflate/brotli	`app.use(compression())`
response-time	Agregar header `X-Response-Time` con la duración de la respuesta	Medir latencia de API
static	Servir archivos estáticos (HTML, CSS, imágenes, JS)	`app.use(express.static('public'))`
serve-favicon	Manejar el favicon de la aplicación	`app.use(favicon(__dirname + '/public/favicon.ico'))`
serve-index	Generar un índice de directorio automáticamente	Mostrar lista de archivos de `/ftp`
vhost	Montar apps Express bajo diferentes virtual hosts	`vhost('api.misitio.com', apiApp)`
morgan	Logger HTTP para desarrollo y producción	`app.use(morgan('dev'))`
errorhandler	Manejar errores mostrando stack trace (solo desarrollo)	`app.use(errorhandler())`

Mejores Prácticas - Versionado

- Como en toda aplicación de escritorio o móvil, en nuestra API habrá mejoras y nuevas funcionalidades.
- Por tanto, es buena práctica ***versionar nuestra API***.
- Las ventajas son:
  - **Principal:** podemos trabajar en nuevas características en una nueva versión mientras los clientes todavía usan la versión actual y los servicios no se verán invalidados ante cambios importantes.
  - **Secundarias:**
    - No obligamos a los clientes (aplicaciones web front-end o mobile) a utilizar la nueva versión de inmediato. Pueden usar la versión actual y migrar a la nueva versión les sea posible.
    - Las versiones actual y nueva básicamente se ejecutan en paralelo y no se afectan entre sí.
- Sobre cómo aplicar el versionado existen varias opiniones: https://stackoverflow.com/questions/389169/best-practices-for-api-versioning

Mejores Prácticas (2)

- **Nombres de Recurso en Plural**
  - No debemos perder de vista que nuestra API será utilizada por humanos y debemos minimizar los malos entendidos o errores de interpretación.
  - Podemos imaginar que cada recurso es un contenedor de una colección de elementos: **estudiantes, jugadores, automóviles**, etc.
  - Nombrar los recursos en plural es una gran ventaja porque da a entender que se trata de una colección de elementos.
- **Evitar nombres de Recurso con verbos**
  - Evitar **`buscarTodosJugadores`**, **`eliminarJugadorPorId`**
  - Dificultan la legibilidad y la búsqueda de las operaciones cuando el proyecto crece.
  - Evita usar verbos en URI, mayúsculas mixtas o extensiones como **`.json`**.
  - Usar guiones medios para separar palabras (**`car-invoice`** en lugar de **`carInvoice`** o **`car_invoice`**).

Mejores Prácticas (3)

- **Aceptar y responder con información en formato JSON**
  - **JSON** es un estándar de facto para las solicitudes y respuestas de una **API Rest**.
  - Si bien solemos asociar **JSON** a **JavaScript** todos los lenguajes modernos tienen mecanismos para procesar datos en este formato.
  - Respecto de enviar datos en formato **JSON** no hay mayores inconvenientes si usamos **`res.json()`** sin embargo para procesar los datos vamos a necesitar utilizar algún **middleware** como **`express.json()`.**
  - La estructura de respuestas JSON debe ser consistente. Por tanto debemos adoptar un estilo unificado. Ejemplos: json:api, Twitter, Facebook.

```
//Según Twitter dentro de un array
[
  {
    "id":1,
    "title":"title"
  },
  {
    "id":2,
    "title":"title"
  }
]
```

```
/*Según json:api*/
{
    "products": [{
        "id": 1,
        "title": "title"
    }]
}
```

```
//Según Twitter y Facebook para un elemento
{
  "id":1,
  "title":"title"
}
```

```
//Según Facebook dentro de un array
{
  "data":[
    {
      "id": 1,
      "title": "title"
    },
    {
      "id": 2,
      "title": "title"
    }
  ]
}
```

Mejores Prácticas (4)

- **Responder con códigos de error HTTP estándar**
  - Algunos ejemplos son:
    - Código **`200`** - Status OK: Cuando hacemos una búsqueda utilizar
    - Código **`201`** – Creado: Cuando agregamos una entidad.
    - Código **`400`** – Error cliente: Cuando la solicitud del cliente es incorrecta.
    - Código **`500`** – Error interno del servidor.
  - Lista completa: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
- **Agrupar recursos lógicamente asociados:**
  - Cuando diseñamos la **API**, puede haber casos en los que tengamos recursos asociados con otros.
  - Es una buena práctica agruparlos en un **endpoint** y anidarlos correctamente.
  - Si se nos presenta el caso **`automoviles/:id/propietarios`** la mejor solución es crear un controlador aparte de **`automoviles`** para **`propietarios`** siendo buena práctica que las **URLs** queden anidadadas.

Mejores Prácticas (5)

- **Integrar filtrado, paginación y ordenación**
  - Si la cantidad de elementos con los que tenemos que tratar es muy grande probablemente debamos implementar una solución que permita:
  - **Filtrar:** es devolver un conjunto menor de elementos que el original según cumplan una condición.
  - **Paginación:** divide los elementos en múltiples "páginas" donde cada página sólo consta de un número limitado de elementos
  - **Ordenación:** es disponer los elementos en un orden según un criterio.
  - Las tres tareas son actividades en las que se desempeñan muy bien los motores de base de datos.
  - En general no las deberíamos hacer en **Node.** En el caso de **MySQL** podemos combinar `SELECT`, `LIMIT`, `OFFSET`
- **Buenas prácticas de seguridad**
  - SSL
  - Autenticación / Autorización

Mejores Prácticas (6)

- **Usar caché para mejorar la performance:** El uso de caché puede mejorar significativamente el rendimiento y reducir la carga en la base de datos. En este sentido existen varias opciones:
  - ***Redis***: se trata de una base de datos en memoria que utiliza almacenamiento basado en ***tablas de hashes*** (pares ***clave-valor***). Opcionalmente permite persistir los datos siendo utilizada como una base de datos durable. Esta solución exige instalar **Redis**.
  - **Caché en memoria:** alternativa a **Redis** utilizando paquetes como **`node-cache`** ó **`memory-cache`** solución para aplicaciones pequeñas o de poco tráfico.
  - **Caché de rutas:** middleware para cachear respuestas a nivel de rutas específicas. Usaremos **`apicache`** para hacer esto fácilmente:

```
import apicache from 'apicache';

let cache = apicache.middleware;

app.get('/api/data', cache('5 minutes'), (req, res) => {
        // Lógica para manejar la solicitud
        const data = actors.findAll();
        res.json(data);
});
```

  - **Headers HTTP para cacheado en clientes:** Indicamos al navegador del cliente que el recurso seguirá siendo el mismo por un lapso de tiempo por lo cual no es necesario volverlo a descargar.

```
res.set('Cache-Control', 'public, max-age=3600'); // 1 hora de cache en el cliente
```

Mejores Prácticas (7)

- **Documentar la API:**
  - Es fundamental porque brinda a los desarrolladores una guía clara de uso.
  - Debe detallar endpoints disponibles, parámetros, formatos de entrada y salida, y posibles errores.
  - Facilita la integración rápida, reduce malentendidos, mejora el mantenimiento y asegura que distintos equipos o aplicaciones puedan interactuar con la API de manera consistente y eficiente.
  - **OpenAPI** es una especificación para describir **APIs REST**.
    - Define cómo éstas deben ser estructuradas y documentadas, lo que permite entender, probar y consumir una **API** de manera estándar, sin tener que revisar el código fuente o la implementación.
    - Una **API** descrita con **OpenAPI** tiene una estructura que incluye endpoints, métodos HTTP (**`GET`, `POST`, `PUT`, `DELETE`**, etc.), parámetros, respuestas, y cualquier otra información relevante.
    - La especificación **OpenAPI** permite definir las **APIs** en un archivo (en formato JSON o YAML), lo que facilita su lectura y comprensión. La versión más utilizada es **OpenAPI 3.x.**
    - **Swagger** fue el proyecto original que implementó una especificación para describir **APIs REST**, la cual más tarde se convirtió en **OpenAPI**.
    - Hoy en día, **Swagger** se refiere a un conjunto de herramientas para trabajar con la especificación **OpenAPI**.

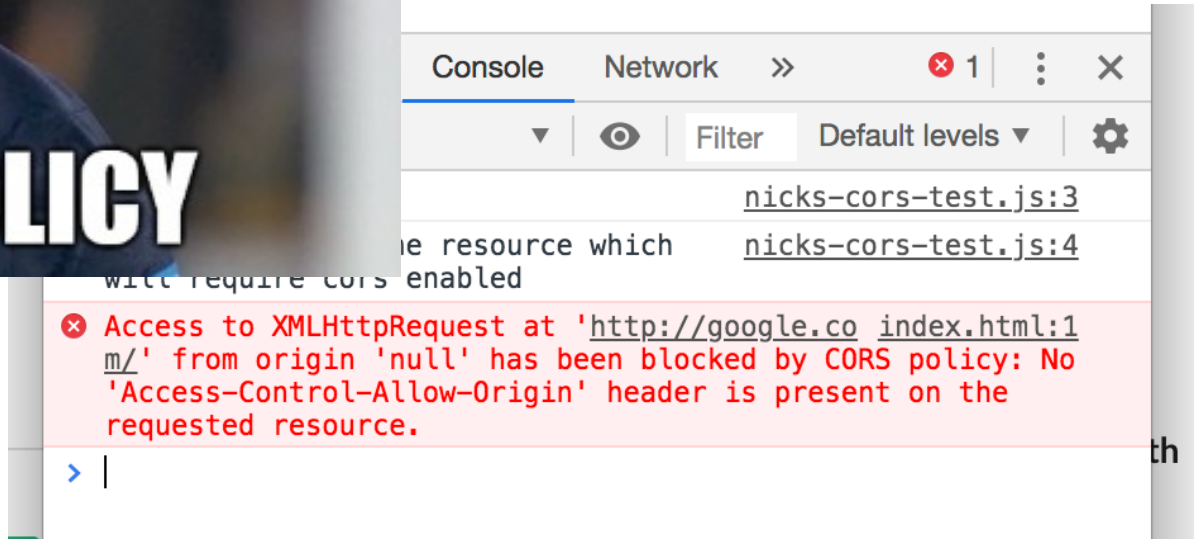
Cuestiones de Seguridad

- Autenticación / Autorización.
- Expiración de Tokens.
- Inyección SQL / NoSQL.
- Validación de datos de entrada **`(express-validator)`**.
- Validar tipos de contenido de las solicitudes.
- Establecer el tamaño máximo de solicitudes.
- Dejar registro de la actividad de la API. **`(morgan + winston)`**
- Tomar precauciones contra los ataques de fuerza bruta **`(express-bouncer)`**
- Usar encabezados HTTP que aumenten la seguridad **`(helmet)`**.
- CORS.

Helmet

- ***Helmet*** está compuesto por un conjunto de pequeñas funciones que ajustan cabeceras de seguridad.
- Algunas de las más importantes son:
  - **`helmet.contentSecurityPolicy():`** restringe qué recursos (scripts, estilos, imágenes) puede cargar el navegador. Previene inyecciones de JavaScript (Cross-Site Scripting).
  - **`helmet.xssFilter():`** Activa el filtro XSS básico del navegador.
  - **`helmet.frameguard():`** Envía la cabecera **`X-Frame-Options`** para evitar que la aplicación se cargue dentro de un **`<iframe>`** (Clickjacking).
  - **`helmet.noSniff():`** Previene que el navegador intente adivinar el tipo de archivo (MIME sniffing).
  - **`helmet.hidePoweredBy():`** Elimina la cabecera **`X-Powered-By: Express.`**
  - **`helmet.hsts():`** Fuerza a los navegadores a usar HTTPS con **`HTTP Strict Transport Security`**.
  - **`helmet.dnsPrefetchControl():`** Controla el DNS prefetching para reducir exposición de información.

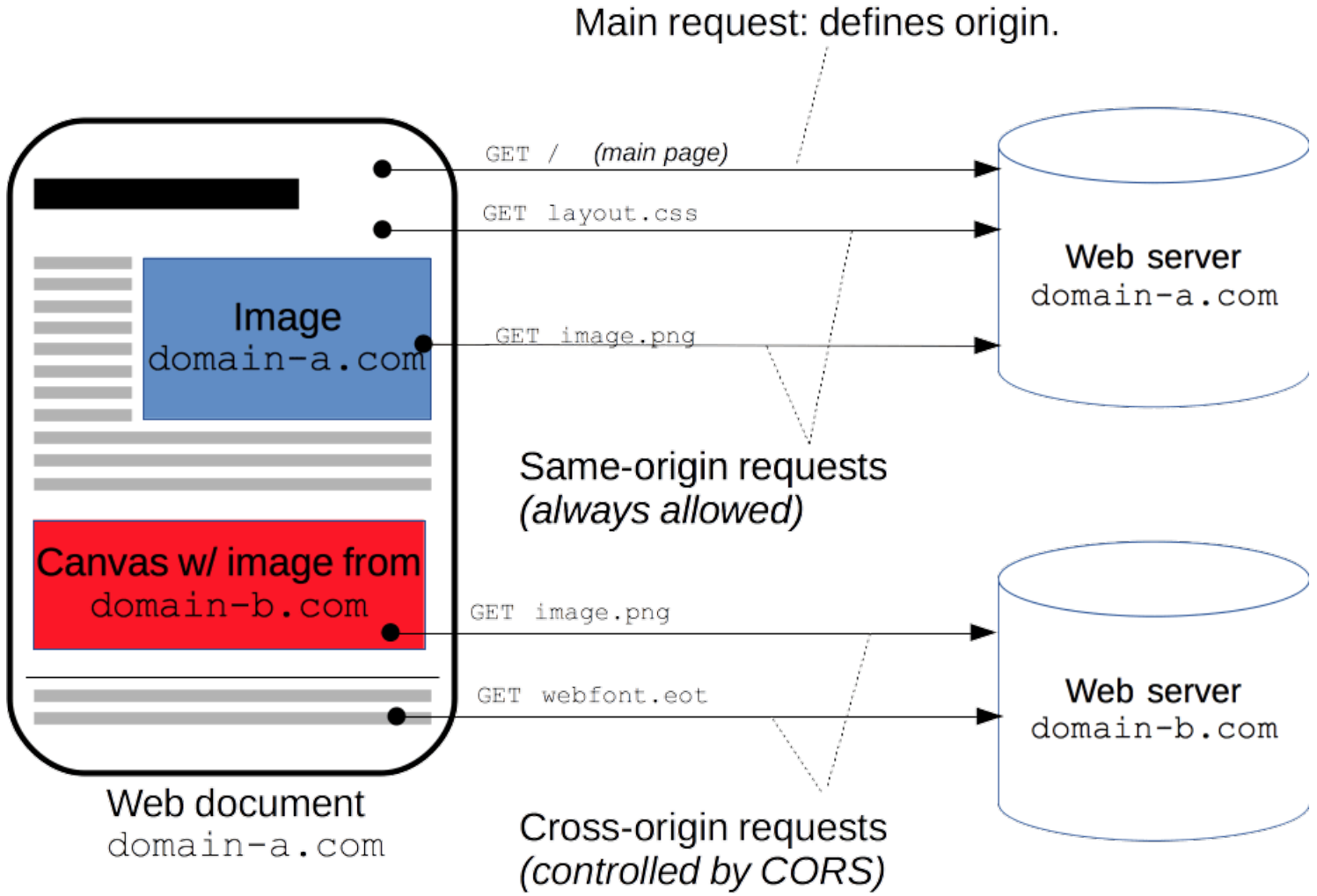
CORS - Cross-Origin Resource Sharing



CORS - Cross-Origin Resource Sharing

- ***Cross-Origin Resource Sharing (CORS)*** es un mecanismo basado en **encabezados HTTP** que permite a un servidor indicar cualquier origen (dominio, esquema o puerto) **distinto del suyo** desde el cual **un navegador debería permitir la carga de recursos**.
- Por ejemplo: el código **JavaScript** de front-end descargado desde `https://domain-a.com` usa `fetch` para realizar una solicitud de `https://domain-b.com/data.json`.
- Por razones de seguridad, los navegadores restringen las **solicitudes HTTP** de origen cruzado iniciadas desde scripts. Por ejemplo, **XMLHttpRequest** y **Fetch API** siguen la política del mismo origen.
- Esto significa que una aplicación web que utiliza esas API solo puede solicitar recursos del mismo origen desde el que se cargó la aplicación, a menos que la respuesta de otros orígenes incluya los encabezados **CORS** correctos.
- **¿Qué solicitudes utilizan CORS?**
  - Este estándar de uso compartido entre orígenes puede permitir solicitudes **HTTP** entre orígenes para:
  - Invocaciones de las API **XMLHttpRequest** o **Fetch**, como se analizó anteriormente.
  - Fuentes web (para el uso de fuentes entre dominios en `@font-face` dentro de **CSS**).
  - Texturas **WebGL**.
  - Imágenes/frames de vídeo dibujados en un `canvas` usando `drawImage()`.
  - Formas CSS a partir de imágenes.

CORS - Cross-Origin Resource Sharing (2)



Bibliografía

- **Libro:** Randy Connolly, Ricardo Hoar. ***“Fundamentals of Web Development, Global Edition”***. 3era Edition. Ed. Pearson. 2022.
- **Libro:** Ethan Brown. ***"Web Development with Node and Express"***. O’Reilly Media, Inc. 2020.
- **Libro:** Simon Holmes, Clive Harber. ***"Getting MEAN"***. 2da Edición. Manning. 2019.
- **Libro**: Luke Welling, Laura Thomson. ***“PHP and MySQL Web Development”***. 5Ta Edición. Addison-Wesley. 2016.
- **Web**: Lokesh Gupta. ***“REST Architectural Constraints”***. REST API Tutorial. Enlace
- **Web**: Jean-Marc Möckel. ***“REST API Design Best Practices Handbook – How to Build a REST API with JavaScript, Node.js, and Express.js”***. FreeCodeCamp. Enlace

Bibliografía (2)

- **Web:** Diogo Souza. ***"Documenting your Express API with Swagger"***. LogRocket. Enlace
- **Web:** Lucila Armentano. ***"Buenas prácticas para el Diseño de una API RESTful Pragmática"***. El Baúl del programador. Enlace.
- **Web:** ***"REST Security Cheat Sheet"***. OWASP Cheat Sheet Series. Enlace.
- **Web:** ***"Nodejs Security"***. OWASP Cheat Sheet Series. Enlace.