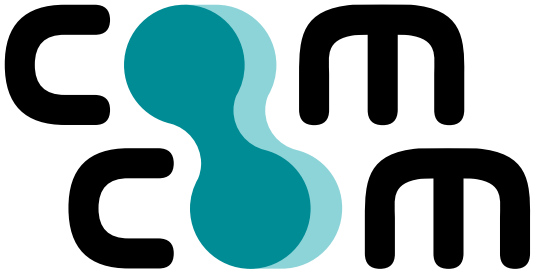
Taller de APIs (RESTful)

ComCom

Departamento de Computación
FCEyN UBA

Primer Cuatrimestre 2026



Introducción a las APIs

¿Qué es una API, y para qué sirve?

¿Cómo se comunican dos programas que no se conocen?

[1]Visual Studio Code, Zed, los de JetBrains, Neovim, emacs, etc.

[2]Claude, Codex, Gemini, etc.

¿Cómo se comunican dos programas que no se conocen?

Las API (Application Programming Interface) son mecanismos que permiten a dos **componentes** de **software comunicarse** entre sí mientras ambos sistemas **cumplan** con un **contrato/especificación**.

[1] Visual Studio Code, Zed, los de JetBrains, Neovim, emacs, etc.

[2] Claude, Codex, Gemini, etc.

¿Cómo se comunican dos programas que no se conocen?

Las API (Application Programming Interface) son mecanismos que permiten a dos **componentes** de **software comunicarse** entre sí mientras ambos sistemas **cumplan** con un **contrato/especificación**.

Ejemplos

- El lector de tarjetas cuando pagás en un local se comunica con la API de la pasarela de pagos.
- Tu IDE[1] favorito habla mediante una API con tu IA/Agente/LLM[2] de confianza.

[1] Visual Studio Code, Zed, los de JetBrains, Neovim, emacs, etc.

[2] Claude, Codex, Gemini, etc.

Tipos de APIs

Existen varios tipos de APIs, algunas son:

- RPC
- WebSocket
- **RESTful**

Siendo estas últimas las más famosas, y las que nos interesarán en el transcurso del taller.

Existen varios tipos de APIs, algunas son:

- RPC
- WebSocket
- **RESTful**

Siendo estas últimas las más famosas, y las que nos interesarán en el transcurso del taller.

¿Qué es una API RESTful?

Las APIs RESTful son un tipo de API que sigue los principios de diseño de REST (Representational State Transfer).

- Utilizan HTTP para realizar operaciones CRUD (Create, Read, Update, Delete) sobre recursos.
- **Un recurso es cualquier entidad que se pueda identificar, como un usuario, un producto, una publicación, etc.**
  - Se puede pensar *vagamente* como «información almacenada identificable».

Para entender como funcionan las APIs RESTful, es importante conocer los métodos HTTP más comunes:

¿Cómo devuelven la información?

Las APIs RESTful suelen devolver la información en formato JSON, aunque también pueden usar XML u otros formatos. A su vez, utilizan códigos de estado HTTP.

JSON

```json
{
  "id": 1,
  "name": "Juan Pérez",
  "email": "jperez@dc.uba.ar",
  ...
}
```

```
HTTP/1.1 200 OK
```

¿Cómo devuelven la información?

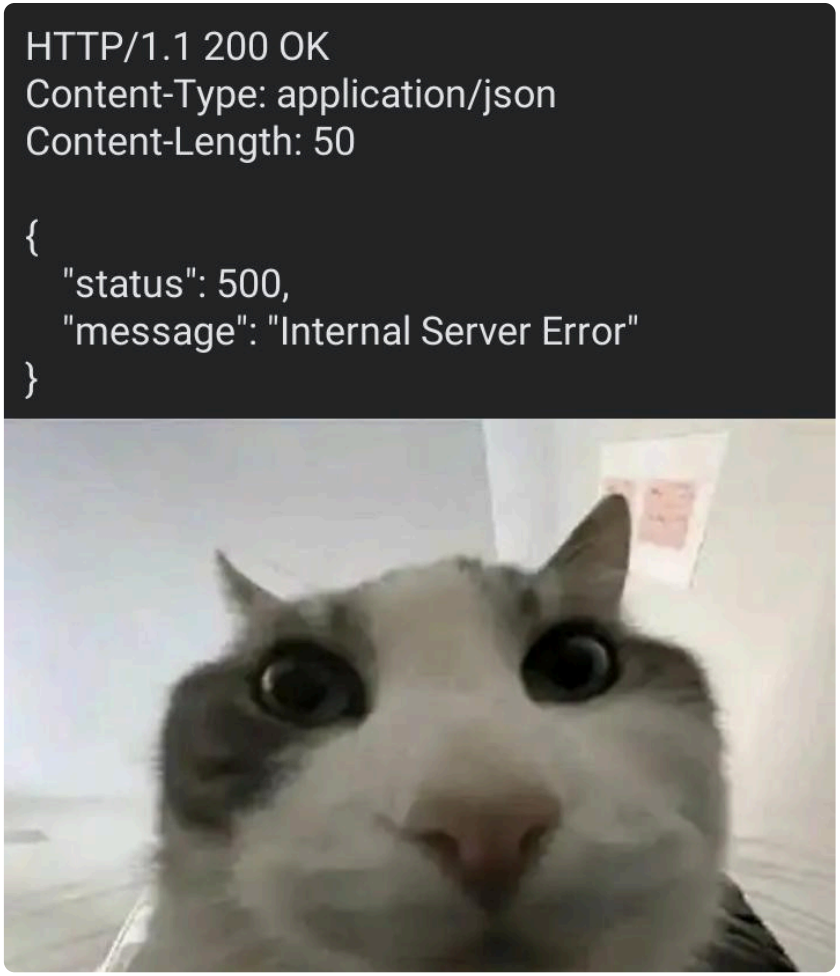
Es importante remarcar que **no toda request devuelve contenido** en el cuerpo.

Por ejemplo, una request de eliminación exitosa del verbo `DELETE` suele devolver un código de estado `204 No Content`, indicando que la operación fue exitosa pero no hay contenido para devolver.

¿Cómo devuelven la información?

Es importante remarcar que **no toda request devuelve contenido** en el cuerpo.

Por ejemplo, una request de eliminación exitosa del verbo `DELETE` suele devolver un código de estado `204 No Content`, indicando que la operación fue exitosa pero no hay contenido para devolver.



También está bueno **ser consistentes** con los códigos de error y **no devolver mensajes redundantes**.

Códigos de estado HTTP

Los códigos de estado HTTP son respuestas estándar que indican el resultado de la solicitud realizada a la API.

- 1xx: Informativos
- 2xx: Éxito
- 3xx: Redirección
- 4xx: Error del cliente
- 5xx: Error del servidor

Headers HTTP

Los headers (encabezados) son **metadatos** que viajan junto a la request (cliente → servidor) o la response (servidor → cliente).

Contienen información adicional que no va en el body.

Headers HTTP

Los headers (encabezados) son **metadatos** que viajan junto a la request (cliente → servidor) o la response (servidor → cliente).

Contienen información adicional que no va en el body.

```
1 GET /me HTTP/1.1
2 Host: api.ejemplo.com
3 Authorization: Bearer eyJhbGciOiJIUz...
4 Content-Type: application/json
```

Los headers (encabezados) son **metadatos** que viajan junto a la request (cliente → servidor) o la response (servidor → cliente).

Contienen información adicional que no va en el body.

```
1 GET /me HTTP/1.1
2 Host: api.ejemplo.com
3 Authorization: Bearer eyJhbGciOiJIUz...
4 Content-Type: application/json
```

Algunos headers comunes:

- `Content-Type`: indica el formato del body (p. ej. `application/json`).
- `Authorization`: información de autenticación y autorización.

Hay varias formas de interactuar con una API RESTful, algunas de las más comunes son:

- Desde la terminal usando herramientas como `curl` o `httpie`.
- Usando clientes HTTP como Postman o Insomnia.
- Usando la propia documentación de la API, si esta tiene una interfaz interactiva (como Swagger UI o Scalar).

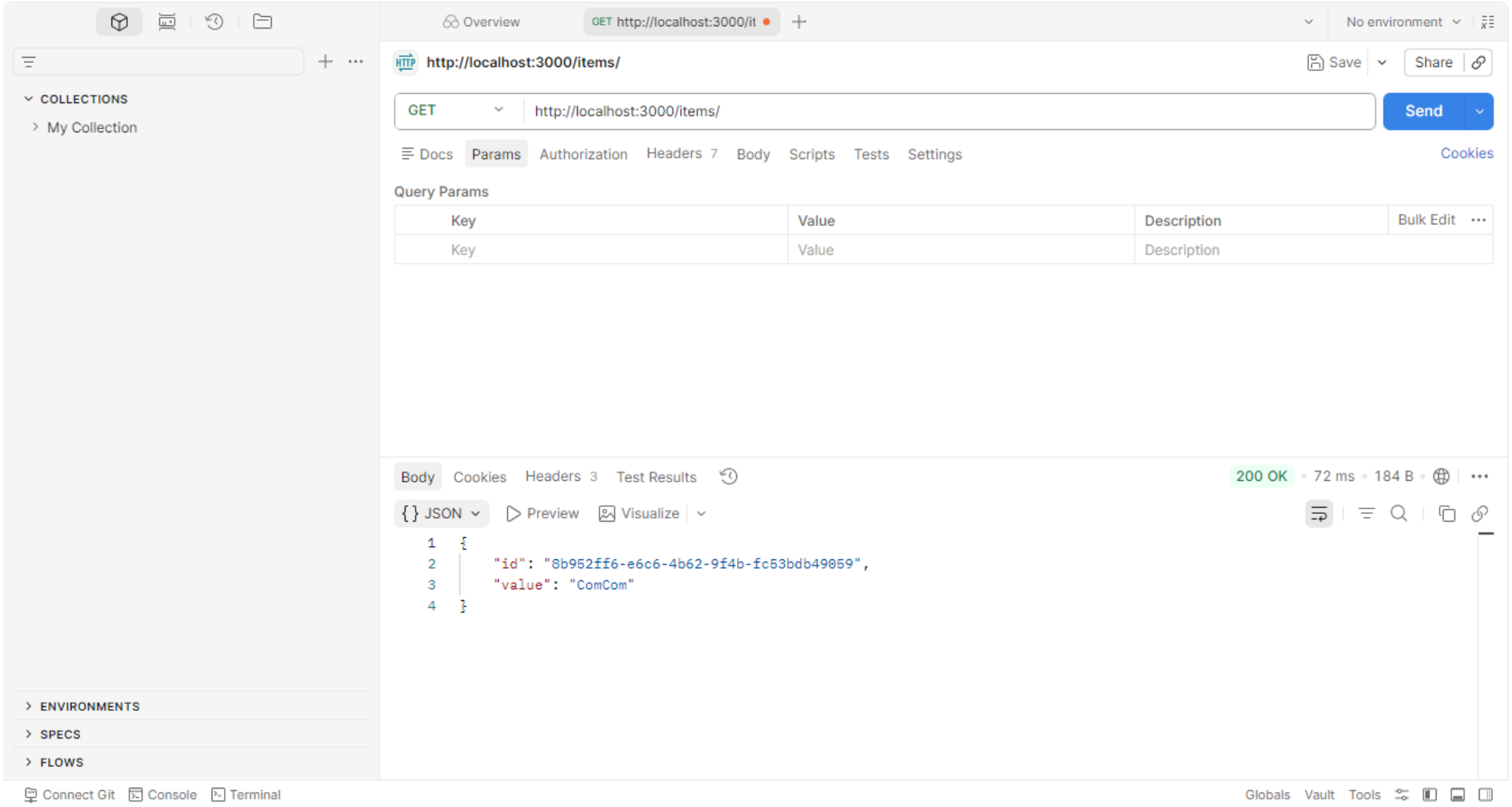
Terminal (p. ej.: curl y httpie)

```
λ ~/ curl http://localhost:3000/items/ \
  --header 'Accept: application/json, multipart/form-data, text/plain'
{"id":"dafe32f6-fcd3-45d3-9fec-ca7ae0fcdee9","value":"ComCom"}%
```

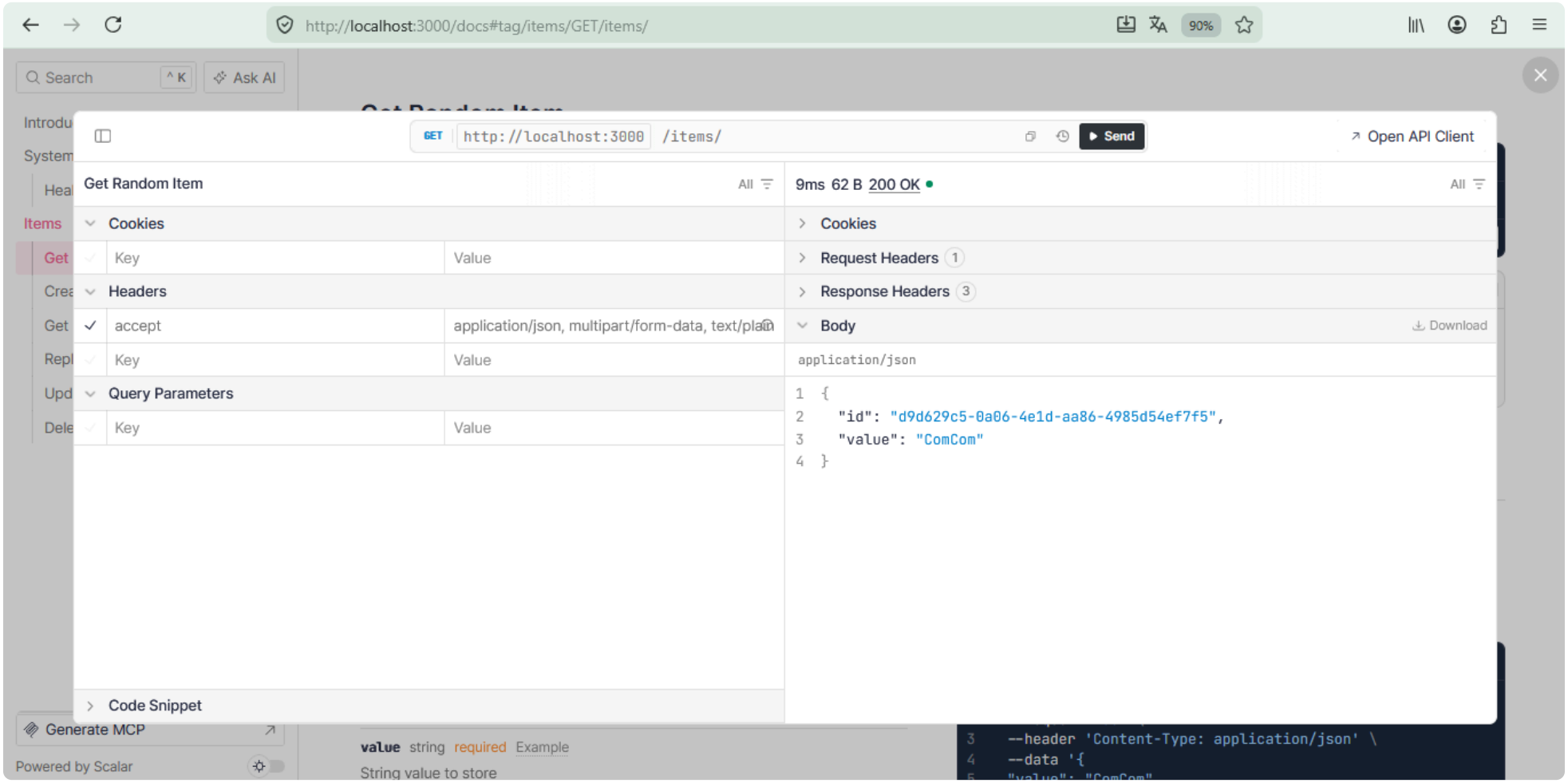
```
λ ~/ http GET http://localhost:3000/items/ \
  Accept:'application/json, multipart/form-data, text/plain'
HTTP/1.1 200 OK
Content-Length: 62
Content-Type: application/json;charset=utf-8
Date: Tue, 21 Apr 2026 17:49:55 GMT

{
    "id": "dafe32f6-fcd3-45d3-9fec-ca7ae0fcdee9",
    "value": "ComCom"
}
```

Clientes HTTP (p. ej.: Postman)



Documentación interactiva (p. ej.: Scalar)



Interactuando con una API RESTful

Ejercicio 1: Strings anónimos

Levantamos un servidor, este expone una API RESTful lista para usar. La idea de este ejercicio es familiarizarse con alguna forma para interactuar con la API (terminal, cliente HTTP o documentación interactiva) y probar los distintos endpoints que expone la API.

Esta API RESTful expone:

- `/items`: permite solicitudes `GET` y `POST`.
- `/items/{id}`: permite solicitudes `GET`, `PUT`, `PATCH` y `DELETE`.

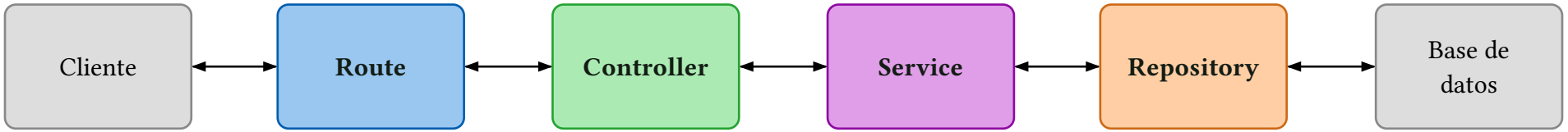
Consignas:

- Entren a **/docs** de la API, donde van a encontrar la documentación interactiva.
- Prueben los distintos verbos HTTP sobre las rutas indicadas y observen las respuestas que devuelve la API.[1]

[1] En el taller usaremos la documentación interactiva, pero pueden usar cURL o lo que prefieran.

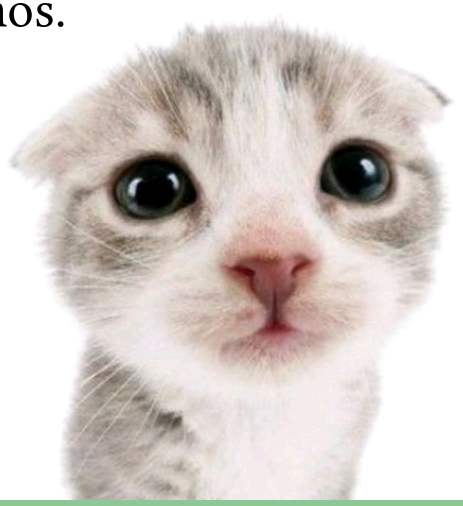
¿Cómo se estructura un proyecto?

Flujo de una request



- La **route** recibe la request y la dirige al **controller**.
- El **controller** decide qué hacer y delega al **service**.
- El **service** ejecuta la lógica de negocio y usa el **repository** para acceder a los datos.
- Los **schemas** validan los datos automáticamente (los vamos a ver más adelante).

Vamos a ir viendo cada una de estas partes a medida que las necesitemos.



Una forma de organizar el código es separarlo en directorios, cada uno con una responsabilidad:

```
src/
├── routes/          Definen los endpoints (paths + verbos HTTP)
├── controllers/     Reciben la request y deciden la respuesta
├── services/        Contienen la lógica de negocio
├── repositories/    Acceso a la base de datos
├── schemas/         Definen la forma (y validación) de los datos
├── middlewares/     Funciones que interceptan requests
└── databases/       Configuración/conexión a la base de datos
```

También puede estar presente un directorio `utils` para funciones de utilidad.

Generalmente, para cada recurso (p. ej. `items`, `users`, `auth`) existe un archivo en `routes`, `controllers` y `services`; a veces también en `repositories`.

Dependiendo del proyecto, las tecnologías utilizadas o las preferencias del equipo, algunos directorios pueden no existir o pueden tener otros nombres.

Rutas / Routes / Endpoints

Definición

Una ruta (también conocida como endpoint) es una URL que representa un recurso o una colección de recursos en una API RESTful.

- Las rutas se utilizan para acceder a los recursos y realizar operaciones sobre ellos utilizando los verbos HTTP.

Ejemplo

En el ejercicio de los strings anónimos, las rutas expuestas por la API RESTful son:

Ejemplo

En el ejercicio de los strings anónimos, las rutas expuestas por la API RESTful son:

- `/items`: representa la colección de ítems y permite realizar operaciones como obtener un ítem aleatorio de la lista (`GET`) o crear un nuevo ítem (`POST`).

Ejemplo

En el ejercicio de los strings anónimos, las rutas expuestas por la API RESTful son:

- `/items`: representa la colección de ítems y permite realizar operaciones como obtener un ítem aleatorio de la lista (`GET`) o crear un nuevo ítem (`POST`).
- `/items/{id}`: representa un ítem específico identificado por su `id` y permite realizar operaciones como obtener los detalles de un ítem (`GET`), actualizar un ítem (`PUT` o `PATCH`) o eliminar un ítem (`DELETE`).

Route parameters, query parameters, body & response

- Route parameters (p. ej., `{id}`): se utilizan para identificar recursos específicos.
  - `/users/:id`
- Query parameters: se pueden utilizar, por ejemplo, para filtrar, ordenar o paginar recursos.
  - `/users?age=30&sort=asc&page=2`
- Body: se utiliza para enviar datos al servidor en solicitudes `POST`, `PUT` o `PATCH`.
  - `{ "name": "Juan Pérez" }`
- Response: es la información que el servidor devuelve al cliente después de procesar una solicitud.
  - `{ "id": 1, "name": "Juan Pérez" }`
  - `HTTP/1.1 200 OK`

En el template tienen una API con varias rutas ya definidas, por ejemplo:

```TypeScript
export const ejemploRoutes = new Elysia()
  .get("/ejemploGet", ejemploController.ejemploRecursoGet)
  .post("/ejemploPost", ejemploController.ejemploRecursoPost);
```

Consigna:

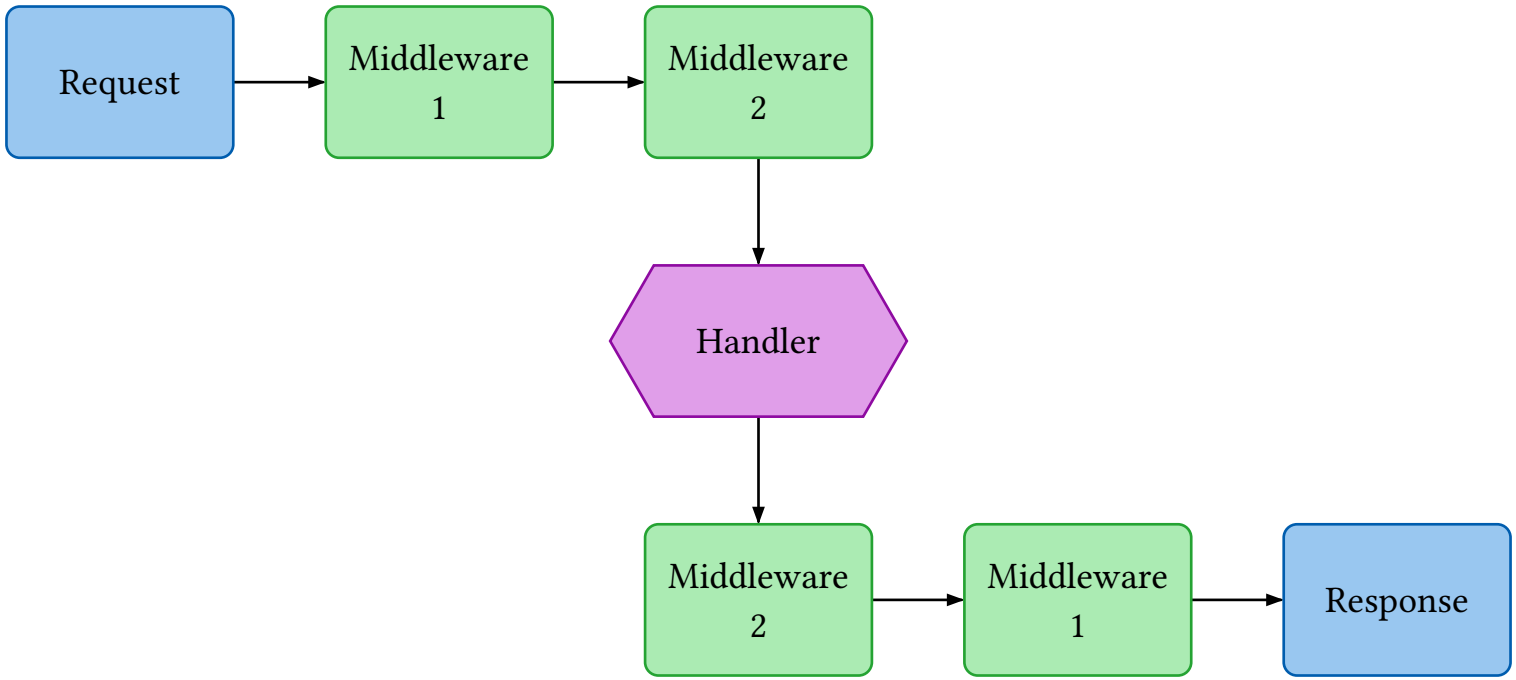
1. Descárguense en template y ábranlo en su editor de código.
2. Hagan **`bun install`** para instalar las dependencias.
3. Abran **`auth.routes.ts`** ubicado en la carpeta `src/routes`.
4. Registren dos rutas de tipo `POST`:
   - `/auth/register` → `authController.register`
   - `/auth/login` → `authController.login`
5. Hagan **`bun dev`** para levantar la API.
6. Verifiquen que las rutas aparezcan en **`http://localhost:3000/docs`**.

Middlewares

¿Qué es un middleware?

Un middleware es una función que se ejecuta **antes** y/o **después** de que un handler (controller) procese una request.

Se puede pensar como una «capa» que intercepta las solicitudes y respuestas.



¿Para qué sirven?

Los middlewares permiten agregar funcionalidad a la aplicación sin modificar los handlers individualmente. Por ejemplo:

- **Logging**: registrar cada solicitud que llega al servidor.
- **Autenticación**: verificar que el usuario tenga un token válido.
- **Autorización**: verificar que el usuario tenga permiso para acceder a la ruta.
- **Validación**: verificar que los datos de la solicitud sean correctos.

Middlewares en Elysia

```typescript
import { Elysia } from 'elysia'

export function middleware_ejemplo(app: Elysia) {
  let middleware_app = app
    .onBeforeHandle(({ request, set }) => {
      // Se ejecuta antes de llegar al handler
    })
    .onAfterHandle(({ request, set }) => {
      // Se ejecuta después de llegar al handler
    });

  return middleware_app;
}
```

Hay más métodos que `.onBeforeHandle` y `.onAfterHandle`, por mencionar uno: `.derive` permite agregar contexto a la request (como por ejemplo, el tiempo exacto en que el cliente mandó la request).

¿Cómo se usa un middleware en Elysia?

Para usar un middleware, simplemente se agrega con `.use()`:

TypeScript

```ts
import { Elysia } from "elysia";
import { logger } from "./middlewares/logger";
import { authenticator } from "./middlewares/authenticator";

const app = new Elysia()
  .get("/sin-middleware", () => "No uso ningún middleware.")
  .use(logger)
  .get("/con-logger", () => "Uso logger.")
  .use(authenticator)
  .get("/secret", () => "Uso logger y luego authenticator.")
```

Ejercicio 3: Proteger rutas con autenticación

En el template tienen las rutas **`/secret`** y **`/users`** sin protección...

Consigna:

1. Accedan a **`http://localhost:3000/docs`**.
2. Mándenle un `GET` a **`/secret`** y **`/users`**, vean qué devuelven.
3. Abran los archivos **`secret.routes.ts`** y **`users.routes.ts`** ubicados en la carpeta **`src/routes`**.
4. Importen el middleware **`authenticator`** de **`../middlewares/authenticator`**.
5. Agreguen **`.use(authenticator)`** antes de la definición de la ruta.
6. Prueben acceder a **`/secret`** y **`/users`**: deberían recibir un status code: `400` y en el body: `Falta el token de autenticación`.

Autenticación

¿Cómo sabe la API quién sos?

Las APIs RESTful son **stateless**: no recuerdan quién sos entre request y request.

¿Cómo sabe la API quién sos?

Para resolver esto, se usa un esquema de **tokens**:

1. De una forma u otra, obtenés un **token de acceso** (access token).
2. Luego, en cada request que requiera autenticación, el cliente envía ese token en el header `Authorization`:
   - `Authorization: Bearer <token>`

¿Cómo sabe la API quién sos?

Para resolver esto, se usa un esquema de **tokens**:

1. De una forma u otra, obtenés un **token de acceso** (access token).
2. Luego, en cada request que requiera autenticación, el cliente envía ese token en el header `Authorization`:
   - `Authorization: Bearer <token>`

En nuestra API el **token de acceso** se obtiene mediante el endpoint `POST /auth/login`.

1. El cliente envía sus credenciales (usuario y clave) al endpoint de login.
2. Si son correctas, el servidor devuelve un **token de acceso**.

JWT (JSON Web Token)

El tipo de token más común es el **JWT** (JSON Web Token)[1].

Lo único que interesa para este taller, es que el token es un string que le permite al servidor identificar al usuario, garantizando que no fue alterado.

[1] ¡A pesar de la creencia popular, no es obligatorio usar JWT!

Las contraseñas **nunca se guardan en texto plano**. Se guardan como un **hash**: el resultado de una función de una sola vía.

Texto plano (MAL)

```
1 usuario: valen
2 clave: 123456
```

Hash (BIEN)

```
1 usuario: juani
2 clave: $2b$10$K4f3...
```

Las contraseñas **nunca se guardan en texto plano**. Se guardan como un **hash**: el resultado de una función de una sola vía.

Texto plano (MAL)

```
1 usuario: valen
2 clave: 123456
```

Hash (BIEN)

```
1 usuario: juani
2 clave: $2b$10$K4f3...
```

- Al **registrar**: se hashea la clave antes de guardarla.
- Al **hacer login**: se hashea la clave ingresada y se compara con el hash guardado.
- Si alguien accede a la base de datos, **no puede recuperar las claves originales**.

Controllers, Services y Repositories

¿Qué es un Controller?

Un controller **recibe la request** HTTP, llama al service correspondiente, y **decide qué responder** (código de estado + body).

TypeScript

```ts
export function tweetAnónController(contexto) {
  if (!contexto.body || !contexto.body.textoTweet) {
    set.status = 400;
    return "Falta el body o el texto del tweet";
  }
  return tweetAnónService.publicarTweetAnón(contexto.uuid,
  contexto.body.textoTweet);
}
```

¿Qué es un Controller?

Un controller **recibe la request** HTTP, llama al service correspondiente, y **decide qué responder** (código de estado + body).

```typescript
export function tweetAnónController(contexto) {
  if (!contexto.body || !contexto.body.textoTweet) {
    set.status = 400;
    return "Falta el body o el texto del tweet";
  }
  return tweetAnónService.publicarTweetAnón(contexto.uuid,
  contexto.body.textoTweet);
}
```

- **No contiene lógica de negocio** (eso va en el service).
- Se encarga de interpretar el resultado y elegir el **status code** adecuado.
- Maneja los **errores** que puede lanzar el service.

¿Qué es un Service?

Un service contiene la **lógica de negocio**: aplica reglas, valida campos según las reglas de negocio y usa el **repository** para acceder a los datos.

```typescript
export function publicarTweetAnón(uuid, textoTweet) {
  const user = usersRepository.obtenerPorUuid(uuid);
  if (!user) throw new Error("Usuario no encontrado");
  if (user.baneado) throw new Error("Usuario baneado");
  if (textoTweet.length > 280) throw new Error("El texto del tweet es demasiado largo");
  const tweet = tweetAnónRepository.crear(uuid, textoTweet);
  return tweet;
}
```

¿Qué es un Service?

Un service contiene la **lógica de negocio**: aplica reglas, valida campos según las reglas de negocio y usa el **repository** para acceder a los datos.

```typescript
export function publicarTweetAnón(uuid, textoTweet) {
  const user = usersRepository.obtenerPorUuid(uuid);
  if (!user) throw new Error("Usuario no encontrado");
  if (user.baneado) throw new Error("Usuario baneado");
  if (textoTweet.length > 280) throw new Error("El texto del tweet es demasiado largo");
  const tweet = tweetAnónRepository.crear(uuid, textoTweet);
  return tweet;
}
```

- **No conoce HTTP**: no sabe de requests, responses, ni status codes.
- **No accede a la base de datos directamente**: usa el repository.
- Lanza **errores** que el controller traduce a respuestas HTTP.

¿Qué es un Repository?

Un repository se encarga del **acceso a los datos**: leer, escribir, actualizar y eliminar registros de la base de datos.

- `usersRepository.obtenerPorUuid(uuid)`:

```TypeScript
export function obtenerPorUuid(uuid) {
  // Código que obtiene el usuario de la base de datos usando su UUID
}
```

- `tweetAnónRepository.crear(uuid, textoTweet)`:

```TypeScript
export function crear(uuid, textoTweet) {
  // Código que crea un nuevo tweet en la base de datos con el UUID del usuario y el texto del tweet
}
```

¿Qué es un Repository?

Un repository se encarga del **acceso a los datos**: leer, escribir, actualizar y eliminar registros de la base de datos.

- `usersRepository.obtenerPorUuid(uuid)`:

```typescript
export function obtenerPorUuid(uuid) {
  // Código que obtiene el usuario de la base de datos usando su UUID
}
```

- `tweetAnónRepository.crear(uuid, textoTweet)`:

```typescript
export function crear(uuid, textoTweet) {
  // Código que crea un nuevo tweet en la base de datos con el UUID del usuario y el texto del tweet
}
```

- Es el **único** que habla con la base de datos.
- Expone operaciones **CRUD** (Create, Read, Update, Delete).
- No tiene lógica de negocio ni conoce HTTP.

Controller vs Service vs Repository

	Controller	Service	Repository
¿Conoce HTTP?	✓	×	×
¿Lógica de negocio?	×	✓	×
¿Accede a la DB?	×	×	✓
¿Decide status code?	✓	×	×
¿Valida datos?	Puede[1]	Reglas de negocio	×

[1] Lo ideal es que no tenga que hacerlo o lo haga lo menos posible, ya veremos por qué.

Algunas operaciones **tardan**: leer un archivo, hacer una consulta a la DB, llamar a otra API...

Algunas operaciones **tardan**: leer un archivo, hacer una consulta a la DB, llamar a otra API...

TypeScript/JavaScript **no se queda esperando**. En vez de eso, te devuelve una **promesa**: un objeto que dice *«todavía no tengo el resultado, pero te lo voy a dar cuando esté listo»*.

```typescript
// Esto NO devuelve el usuario, devuelve una PROMESA de un usuario.
const promesa = db.query("SELECT * FROM users WHERE uuid = ?", [uuid]);
```

Algunas operaciones **tardan**: leer un archivo, hacer una consulta a la DB, llamar a otra API...

TypeScript/JavaScript **no se queda esperando**. En vez de eso, te devuelve una **promesa**: un objeto que dice *«todavía no tengo el resultado, pero te lo voy a dar cuando esté listo»*.

```TypeScript
// Esto NO devuelve el usuario, devuelve una PROMESA de un usuario.
const promesa = db.query("SELECT * FROM users WHERE uuid = ?", [uuid]);
```

Para decirle a TypeScript/JavaScript *«esperá a que termine antes de seguir»*, usamos `await`:

```TypeScript
// Ahora sí, esperamos a que la promesa se resuelva y obtenemos el usuario.
const usuario = await db.query("SELECT * FROM users WHERE uuid = ?", [uuid]);
```

Para poder usar `await` dentro de una función, hay que marcarla como `async`:

```typescript
async function obtenerUsuario(uuid) {
  const usuario = await db.query("SELECT * FROM users WHERE uuid = ?", [uuid]);
  return usuario;
}
```

- Si una función es `async`, **siempre devuelve una promesa** (aunque parezca que devuelve un valor directo).
- Si se olvidan de poner `await`, van a recibir el objeto `Promise` en vez del resultado.

Parte A – Service (`auth.service.ts`):

1. Implementen `registrar`: hasheen la clave (con `await hashearClave(clave)`) y creen el usuario (con `crearUsuario(nombreDeUsuario, claveHasheada)`), luego devuelvan su UUID.

2. Implementen `login`: obtengan el objeto usuario (con `obtenerUsuarioPorNombreDeUsuario(nombreDeUsuario)`) y comparen la clave que les pasaron con la clave guardada (con `await compararClaves(clave, usuario.claveHasheada)`), luego devuelvan su UUID.

Parte B – Controller (`auth.controller.ts`):

1. Completen `registrar`: validen que el body exista, luego sus parámetros (que existan, largos mínimos), llamen al service dentro del try/catch y devuelvan el token con el status code `201` o que el usuario ya existe con el status code `409`, según corresponda.

2. Completen `login`: validen que el body exista, luego sus parámetros (que existan), llamen al service dentro del try/catch y devuelvan el token con el status code `200` o que las credenciales son inválidas con el status code `401`, según corresponda.

Ejercicio 4: Implementar login y register

Parte A – Service (`auth.service.ts`):

1. Implementen `registrar`: hasheen la clave (con `await hashearClave(clave)`) y creen el usuario (con `crearUsuario(nombreDeUsuario, claveHasheada)`), luego devuelvan su UUID.

2. Implementen `login`: obtengan el objeto usuario (con `obtenerUsuarioPorNombreDeUsuario(nombreDeUsuario)`) y comparen la clave que les pasaron con la clave guardada (con `await compararClaves(clave, usuario.claveHasheada)`), luego devuelvan su UUID.

Parte B – Controller (`auth.controller.ts`):

1. Completen `registrar`: validen que el body exista, luego sus parámetros (que existan, largos mínimos), llamen al service dentro del try/catch y devuelvan el token con el status code `201` o que el usuario ya existe con el status code `409`, según corresponda.

2. Completen `login`: validen que el body exista, luego sus parámetros (que existan), llamen al service dentro del try/catch y devuelvan el token con el status code `200` o que las credenciales son inválidas con el status code `401`, según corresponda.

Prueben: registrar un usuario, hacer login, intentar con datos inválidos.

Ejercicio 4: Implementar login y register

¿Cuántos `if` tuvieron que escribir para validar los datos?

¿Qué pasa si agregan un campo nuevo al body? ¿Y si cambian un largo mínimo?

Schemas

¿Qué es un Schema?

Un schema **define la forma de los datos**: qué campos tiene el body, qué tipos tienen, qué restricciones cumplen, y qué devuelve la respuesta.

TypeScript

```ts
const bodySchema = t.Object({
  nombreDeUsuario: t.String({ minLength: 3 }),
  clave: t.String({ minLength: 6 }),
});
```

¿Qué es un Schema?

Un schema **define la forma de los datos**: qué campos tiene el body, qué tipos tienen, qué restricciones cumplen, y qué devuelve la respuesta.

TypeScript

```ts
const bodySchema = t.Object({
  nombreDeUsuario: t.String({ minLength: 3 }),
  clave: t.String({ minLength: 6 }),
});
```

Si la request no cumple con el schema, **el framework la rechaza automáticamente** con un `400 Bad Request`, sin necesidad de escribir un solo `if`.

Sin schema (`register` en `auth.controller.ts`), no entra todo el código en la diapositiva:

```typescript
// ... código anterior
  if (!body) {
    set.status = 400
    return "Falta body en la request"
  }
  const { nombreDeUsuario, clave } = body;
  if (!nombreDeUsuario || !clave) {
    set.status = 400
    return "Faltan el nombre del usuario y la clave en la request";
  }
  if (nombreDeUsuario.length < 3) {
    set.status = 400
    return "El nombre de usuario debe tener al menos 3 caracteres";
  }
// ... una validación similar para la clave y el resto del código
```

Con schema (`register` en `auth.controller.ts`), entra todo el código en la diapositiva:

TypeScript

```typescript
const { body, set } = contexto;
const { nombreDeUsuario, clave } = body;

// Acá estarían los chequeos manuales, pero ya no hacen falta porque el schema se encarga de eso.

try {
  const uuid = await authService.register(nombreDeUsuario, clave)

  const tokenDeAcceso = await crearTokenDeAcceso(uuid)
  set.status = 201
  return {tokenDeAcceso}
} catch {
  set.status = 409
  return "Ya existe un usuario con ese nombre de usuario";
}
```

¿Cómo se usa un Schema?

El schema se pasa como **tercer argumento** en la definición de la ruta:

```TypeScript
// Antes (sin schema):
.post("/auth/register", authController.register)

// Después (con schema):
.post("/auth/register", authController.register,
registrarUnUsuarioSchema)
```

¿Cómo se usa un Schema?

El schema se pasa como **tercer argumento** en la definición de la ruta:

```TypeScript
// Antes (sin schema):
.post("/auth/register", authController.register)

// Después (con schema):
.post("/auth/register", authController.register, registrarUnUsuarioSchema)
```

Además, el controller puede tipar sus parámetros con el **contrato** del schema:

```TypeScript
export async function register(
  contexto: Context<RegistrarUnUsuarioRouteContract> // Toma un contexto que cumple con el contrato
): Promise<RegistrarUnUsuarioRouteContract['response'][201 | 409]> // Devuelve 201 o 409, según el contrato
{ ... }
```

Esto da **autocompletado** y **chequeo de tipos** en el editor.

Bonus: documentación automática

Al definir schemas, la documentación interactiva (Swagger / Scalar) muestra **automáticamente**:

- Los campos esperados en el body, con sus tipos y restricciones.
- Las posibles respuestas con sus códigos de estado.
- Ejemplos de uso.

Bonus: documentación automática

Al definir schemas, la documentación interactiva (Swagger / Scalar) muestra **automáticamente**:

- Los campos esperados en el body, con sus tipos y restricciones.
- Las posibles respuestas con sus códigos de estado.
- Ejemplos de uso.

Todo sin escribir documentación a mano. Solo por definir los schemas.

Parte A – Schemas (`auth.schemas.ts`):

1. **Completen el schema** `const iniciarSesionSchema = { ... }`[1]:
   - Body: `nombreDeUsuario` (string), `clave` (string).
   - Response 200: objeto con `tokenDeAcceso` (string).
2. **Completen el contrato** `interface IniciarSesionRouteContract { ... }`[2]:
   - Body: `nombreDeUsuario` (string), `clave` (string).
   - Response 200: objeto con `tokenDeAcceso` (string).
   - Response 401: string (mensaje de error).
3. Modifiquen `auth.routes.ts`: pasen los schemas como tercer argumento a cada `.post()`.
   - `.post("/auth/register", authController.register, registrarUnUsuarioSchema)`
   - `.post("/auth/login", authController.login, iniciarSesionSchema)`

[1] Básense en el schema `const registrarUnUsuarioSchema = { ... }` que ya está implementado.

[2] El contrato `interface RegistrarUnUsuarioRouteContract { ... }` ya está implementado, úsenlo como referencia.

Parte B – Autocompletado y chequeo de tipos (opcional):

1. Cambien la signatura de las funciones en `auth.controller.ts` para que usen los contratos de los schemas.
   - ```
     export async function register(contexto:
     Context<RegistrarUnUsuarioRouteContract>):
     Promise<RegistrarUnUsuarioRouteContract['response'][201 | 409]> { ... }
     ```
   - ```
     export async function login(contexto:
     Context<IniciarSesionRouteContract>):
     Promise<IniciarSesionRouteContract['response'][200 | 401]> { ... }
     ```
2. **Eliminen las validaciones manuales** que ahora son redundantes.
3. Vayan a `/docs` y vean la documentación auto-generada.

¿¿¿¿¿¿Preguntas??????

Tecnologías usadas en el taller

- Bun (runtime): https://bun.sh/docs
- Elysia (framework): https://elysiajs.com/
- TypeScript (lenguaje): https://www.typescriptlang.org/docs/

REST y HTTP

- El origen de los principios REST: Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*
- HTTP response status codes: MDN Web Docs
- HTTP request methods: MDN Web Docs
- ¿Qué es una API REST?: Red Hat

Autenticación y seguridad

- Introducción a JWT: https://jwt.io/introduction
- OWASP API Security Top 10: https://owasp.org/API-Security/
- Argon2 (hashing de contraseñas): Wikipedia

Herramientas

- Scalar (documentación interactiva): https://scalar.com/
- Postman: https://www.postman.com/
- curl: https://curl.se/
- httpie: https://httpie.io/

APIs públicas

Les dejamos una lista de algunas APIs públicas por si les interesa investigar:

- APIs List: https://apislist.com/
- Free APIs: https://free-apis.github.io/
- Transporte Buenos Aires: https://api-transporte.buenosaires.gob.ar/
- MercadoPago: https://www.mercadopago.com.ar/developers/es/reference
- Spotify: https://developer.spotify.com/documentation/web-api

Bibliografía

- Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine. https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf
- Fielding, R. T., & Reschke, J. (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. https://datatracker.ietf.org/doc/html/rfc7231
- Jones, M., Bradley, J., & Sakimura, N. (2015). *JSON Web Token (JWT)*. RFC 7519. https://datatracker.ietf.org/doc/html/rfc7519
- Mozilla Developer Network. *HTTP Reference*. https://developer.mozilla.org/en-US/docs/Web/HTTP
- OWASP Foundation. *API Security Top 10*. https://owasp.org/API-Security/