Documentación: Zod

Uso en ProjectSENSEI — SenseiAPI

¿Qué es Zod?

Zod es una librería de validación de esquemas para JavaScript y TypeScript. Permite definir la forma y las reglas que deben cumplir los datos entrantes antes de que lleguen a la lógica de negocio.

En esta API se usa para validar el cuerpo (req.body) de las peticiones HTTP antes de procesarlas. Si los datos no cumplen el esquema, Zod rechaza la petición con un mensaje de error claro, sin que el código del endpoint tenga que hacerlo manualmente.

Instalación

```
npm install zod
```

Fragmento 1 — Middleware genérico de validación

Archivo: middlewares/validate.js

```
export const validate = (schema) => (req, res, next) => {
  const result = schema.safeParse(req.body)
  if (!result.success) {
    return res.status(400).json({ message: result.error.errors[0].message })
  }
  req.body = result.data
  next()
}
```

¿Cómo funciona?

‘validate’ es una función de orden superior: recibe un esquema de Zod y devuelve un middleware de Express.

‘schema.safeParse(req.body)’ intenta validar el cuerpo de la petición contra el esquema. Si falla, devuelve un objeto con success: false y los errores. Si pasa, devuelve success: true y los datos limpios en result.data.

‘req.body = result.data’ reemplaza el body original con los datos ya validados y transformados por Zod (por ejemplo, aplicando valores por defecto). Luego se llama a next() para pasar al endpoint.

Fragmento 2 — Definición de esquemas

Archivo: schemas/index.js

```
import { z } from 'zod'

// Esquema para registro de usuario
export const registerSchema = z.object({
  username: z.string().min(3, 'El usuario debe tener mínimo 3 caracteres'),
  password: z.string().min(3, 'La contraseña debe tener mínimo 3 caracteres')
})

// Esquema para login
export const loginSchema = z.object({
  username: z.string().min(1, 'El usuario es obligatorio'),
  password: z.string().min(1, 'La contraseña es obligatoria')
})

// Esquema para crear clase
export const claseSchema = z.object({
  title: z.string().min(1, 'El título es obligatorio'),
  description: z.string().optional().default(''),
  date: z.string().refine(d => !isNaN(new Date(d).getTime()), 'Fecha inválida'),
  capacity: z.number().int().min(1, 'La capacidad debe ser mínimo 1')
})

// Esquema para editar clase (todos los campos opcionales)
export const claseEditSchema = claseSchema.partial()
```

Explicación de cada validador usado

Validador	Qué hace
z.string()	El campo debe ser una cadena de texto
z.string().min(3, msg)	Cadena con mínimo 3 caracteres. Si falla, devuelve msg
z.string().optional()	El campo puede estar ausente (undefined)
z.string().optional().default('')	Si no se envía, se usa '' como valor por defecto
z.string().refine(fn, msg)	Validación personalizada: fn debe devolver true para pasar
z.number().int()	El campo debe ser un número entero
z.number().int().min(1, msg)	Entero de mínimo 1. Si falla, devuelve msg
claseSchema.partial()	Crea un nuevo esquema donde todos los campos son opcionales

Fragmento 3 — Uso del middleware en las rutas

Archivo: routes/auth.js

```
import { validate } from '../middlewares/validate.js'
import { registerSchema, loginSchema } from '../schemas/index.js'

router.post('/register', validate(registerSchema), async (req, res) => {
  // Si llega aquí, req.body ya fue validado por Zod
  const { username, password } = req.body
  ...
})

router.post('/login', validate(loginSchema), async (req, res) => {
  const { username, password } = req.body
  ...
})
```

Archivo: routes/clases.js

```
import { validate } from '../middlewares/validate.js'
import { claseSchema, claseEditSchema } from '../schemas/index.js'

router.post('/', authMiddleware, adminMiddleware, validate(claseSchema), async
(req, res) => {
  // title, description, date, capacity ya validados
  ...
})

router.put('/:id', authMiddleware, adminMiddleware, validate(claseEditSchema),
async (req, res) => {
  // Igual pero todos los campos son opcionales
  ...
})
```

¿Por qué se pone antes del handler?

En Express, los middlewares se ejecutan en orden. Al poner validate(schema) antes del async (req, res) => { ... }, Zod intercepta la petición primero. Si los datos no son válidos, responde con error 400 y el handler nunca se ejecuta. Si son válidos, next() pasa el control al handler con req.body ya limpio.