From Manual to Velocity

The Spec-Driven Way

How we turned weeks into seconds with Orval

Presented by

Keyvan Mahmoudi



The Challenge 🦾



Mission

Build a **NestJS BFF** to act as our API gateway



Integration

Connect 4 different
APIs through WSO2
portal



Reality

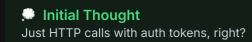
All API calls need authentication and validation

What We Needed to Build



Seemed straightforward...

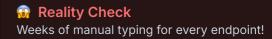
- HTTP clients for each API
- Authentication token management
- Error handling & retries
- Request/response mapping



DTOs 😰

The real nightmare!

- Request body validation schemas
- Query parameter validators
- Path parameter types
- Response type definitions
- Swagger documentation
- Manual maintenance forever...



From Manual Hell... 😂



...to Generated Everything 🐆

Enter Orval 🥕

Input

```
# Just your OpenAPI spec
openapi: 3.0.0
info:
  title: User API
paths:
  /users/{id}:
    get:
      parameters:
        - name: id
          in: path
          schema:
            type: string
      responses:
        '200':
            application/json:
              schema
                $ref: '#/components/schemas
```

Output

- TypeScript types Full type safety
- Zod schemas Runtime validation
- API clients Ready-to-use functions
- React Query hooks Easy data fetching
- Mock handlers Testing with real data

One config file → Everything you need!

Our Orval Configuration

```
export default defineConfig({
 internal: {
   input: './specs/user-service.yaml',
   output: {
      target: './src/generated/internal.ts',
     client: 'axios',
     mock: true,
     override: {
        zod: { enabled: true },
       mutator: {
          path: './src/mutators/internal.ts',
         name: 'internalMutator',
 external: {
    input: './specs/payment.yaml',
```

Multiple Inputs

- YAML and JSON specs
- Separate configs per domain

Rich Output

- Zod schemas with validation
- Mock generation for testing
- Custom mutators for auth



Plot Twist: WSO2 Authentication

Every API call needs authentication tokens



How do we inject custom auth logic into generated code?

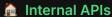
Challenge:

Generated clients can't know about our specific WSO2 requirements

Mutators to the Rescue! 🤵

```
export const internalMutator = async (
  config: AxiosRequestConfig
): Promise<AxiosRequestConfiq> ⇒ {
  // Get internal service token
  const token = await getInternalToken()
  return {
    ... config,
    headers: {
      ...config.headers,
     Authorization: `Bearer ${token}`,
      'Content-Type': 'application/json',
    timeout: 10000,
```

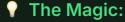
```
export const externalMutator = async (
  config: AxiosRequestConfig
): Promise<AxiosRequestConfiq> ⇒ {
  // Get external partner token
  const token = await getExternalToken()
  return {
    ...config,
    headers: {
      ...config.headers,
      Authorization: `Bearer ${token}`,
      'X-Partner-ID': process env PARTNER_ID,
    timeout: 15000,
```



Auto-injects internal service credentials



Auto-injects external partner credentials



These functions run before EVERY API call automatically

NestJS Integration Made Simple 🗸

```
// src/services/user.service.ts
import { Injectable } from '@nestjs/common
import { getUser, createUser } from '../gene
@Injectable()
export class UserService {
  async findUser(userId: string) {
    return await getUser({ userId })
  async createUser(userData: CreateUserReque
    // Type-safe, auto-validated request
    return await createUser(userData)
```

★ What We Got

- **Auto-authenticated** requests
- Type-safe function calls
- **▼ Runtime validation** with Zod
- Fror handling built-in
- Mock-ready for testing

Before vs After

Before:

HTTP client + auth + types + validation

After:

Import function, call it!

DTOs Without the Pain 😌

The Magic Pipeline

- 1. OpenAPI spec \rightarrow Orval \rightarrow Zod schemas
- T
- 2. Zod schemas \rightarrow nestjs-zod \rightarrow DTOs
- Ŧ
- 3. DTOs \rightarrow NestJS \rightarrow Swagger docs

```
er End Result:
```

WSO2-ready Swagger documentation!

```
// Auto-generated DTO
import { createZodDto } from 'nestjs-zod'
import { CreateUserSchema } from '../generate

export class CreateUserDto extends createZodD
    CreateUserSchema
} }

// Usage in controller
@Post()
async createUser(@Body() dto: CreateUserDto)
// dto is already validated!
return this.userService.createUser(dto)
}
```

- Request validation automatic
- **▼** Swagger docs generated
- Type safety guaranteed

Testing at Lightning Speed 🗲

```
// Auto-generated by Orval
import { rest } from 'msw'
import { getUserMockHandler } from '../gener
// Mock server setup
const server = setupServer(
  getUserMockHandler,
 createUserMockHandler,
  // ... all handlers auto-generated!
it('should fetch user data', async () \Rightarrow {
  // Mock returns realistic data automatical
  const result = await userService.findUser(
  expect(result).toHaveProperty('id')
  expect(result).toHaveProperty('email')
```

Speed Benefits

- No mock setup generated automatically
- Realistic data based on spec examples
- Always in sync regenerate with spec changes

Before vs After

Before:

Write mocks, maintain test data

After:

Import handlers, write tests

The Impact 💥











Frontend: The Journey Continues 2





BFF Built

▼ Services generated ▼ DTOs created **V** Swagger exported



Round Two

BFF Swagger spec → Frontend Orval config → Generate everything!



Frontend Magic

React Query hooks TypeScript types / Zod schemas + mocks

© Complete Loop

External APIs → BFF → Frontend → React App

Frontend Usage in Action of

```
import { useGetUserQuery, useUpdateUserMutation
function UserProfile({ userId }) {
  const { data: user, isLoading, error } = use
  const updateMutation = useUpdateUserMutation
  if (isLoading) return <Spinner />
  if (error) return <ErrorAlert error={error}</pre>
  return (
      <h1>{user.name}</h1>
      <hutton
        onClick=\{() \Rightarrow updateMutation.mutate(
          userId,
          data: { name: 'New Name' }
        Update User
      </button>
    </div>
```

Form Validation

```
// forms/CreateUserForm.tsx
import { CreateUserSchema } from '../api'
import { useForm } from 'react-hook-form'

const form = useForm({
   resolver: zodResolver(CreateUserSchema)
})

const onSubmit = form.handleSubmit((data) ⇒
   // data is fully typed and validated
   createUserMutation.mutate(data)
})
```

What We Get

✓ Automatic loading states
 ♥ Error handling
 ➡ Type-safe forms
 ✓ Realistic test mocks

The Spec-Driven Development Loop 🛟









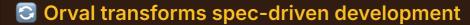






- ▼ No Drift Types always match reality
- ▼ No Bugs Breaking changes caught at compile time
- No Manual Work -**Everything updates** automatically

Key Takeaways 🚀



From OpenAPI specs to complete type-safe clients, automatically

Mutators solve complex auth requirements

Handle any authentication flow with custom request modifiers

Generated mocks accelerate testing

Realistic test data from your actual API specifications

End-to-end automation from API to UI

Complete workflow automation eliminates drift and manual work

Essential Resources 🝃



Orval

orval.dev

nestjs-zod

npmjs.com/package/nestjs-zod

React Query

tanstack.com/query

MSW

mswjs.io

Documentation

- Comprehensive guides
- Real-world examples
- Active communities
- Production-ready

Your Next Steps 💡

Start Small

Pick one API, create your first Orval config, generate a client

Add Authentication

Create mutators for your auth requirements, test with real tokens

Build the Loop

Extend to DTOs, then frontend, establish the automation workflow

Scale & Optimize

Add more APIs, refine your process, enjoy the velocity! 🖋

Thank You! 🎉

Ready to transform your workflow?

Start with **one API** and experience the magic 🐆

Questions?