From OpenAPI *to MCP, in 90 seconds.*

Generate working MCP servers from the OpenAPI specs you already have.

BY SACHIN GUPTA

The boring math that pays for this talk.

200+

REST APIs already documented in OpenAPI

10

MCP servers built

5%

coverage of what agents could call

THE REALISATION

For the common REST API case, an MCP tool needs a useful name, a clear description, a typed input schema, and predictable behavior on success and failure. OpenAPI already gives us most of the raw material.

Anatomy of an OpenAPI operation.

Five things an OpenAPI operation always carries. Four of them have a direct MCP home.

```yaml
paths:
  /pet/{petId}:
    get:
      operationId: getPetById
      summary: Find pet by ID.
      description: Returns a single pet.
      parameters:
        - name: petId
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200': { $ref: '#/.../Pet' }
        '404': { description: Not found }
      security:
        - api_key: []
```

THE FIVE PARTS

- **operationId**
  the unique handle for one HTTP operation
- **summary / description**
  human prose — what the operation does
- **parameters**
  where the inputs live: path, query, header, body
- **responses**
  shape of the success body + every possible error
- **security**
  which credentials the call needs

Anatomy of an MCP tool.

The core surface is name, description, and inputSchema. The protocol carries more — annotations, output schemas, metadata — but these three are what the agent sees first.

ONE TOOL DEFINITION → tools/list

```
{
  "name": "petstore_getPetById",

  "description":
    "Find pet by ID.  Returns a single pet.",

  "inputSchema": {
    "type": "object",
    "properties": {
      "petId": {
        "type": "integer",
        "format": "int64"
      }
    },
    "required": ["petId"]
  }
}
```

ONE INVOCATION → tools/call

```
→  request
{ "method": "tools/call",
  "params": {
    "name": "petstore_getPetById",
    "arguments": { "petId": 10 } } }
```

```
←  response
{ "result": {
    "content": [{
      "type": "text",
      "text": "{ id:10, name: \"doggie\",
                  status: \"sold\" }"
    }],
    "isError": false } }
```

name · description · inputSchema → the three the agent sees first. More fields exist; these are the core surface.

The mapping is mechanical.

Same getPetById on the left, same petstore_getPetById on the right. Colour shows which field becomes which.

OPENAPI 3.x SPEC

```yaml
paths:
  /pet/{petId}:
    get:
      operationId: getPetById
      summary: Find pet by ID.
      description: Returns a single pet.
      parameters:
        - name: petId
          in: path
          schema: { type: integer }
      responses:
        '200': { $ref: Pet }
      security:
        - api_key: []
```

GENERATED MCP TOOL

```jsonc
{
  "name": "petstore_getPetById",
  "description":
    "Find pet by ID. Returns a single pet.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "petId": { "type": "integer" }
    },
    "required": ["petId"]
  }
}

// runtime: HttpPlan · AuthDecorator · ErrorChain
```

operationId → name

summary + description → description

parameters + schema → inputSchema

path · security · responses → runtime

But mechanical isn't trivial.

Six things a naïve auto-mapper gets wrong. Each one, we already handle.

01

$ref resolution

Schemas reference other schemas. Recursively. Sometimes cyclically. The spec is a graph, not a tree.

02

Polymorphic shapes

oneOf, anyOf, allOf — real APIs return different shapes for different cases. Most translators flatten and silently lose information.

03

Description cleanup

OpenAPI descriptions carry markdown, links, 'see the diagram.' Agents read this verbatim, including the parts that don't apply.

04

operationId fallback

Not every operation has one. We compute a deterministic verb + path fallback, sanitised for tool-name character rules.

05

Parameter binding

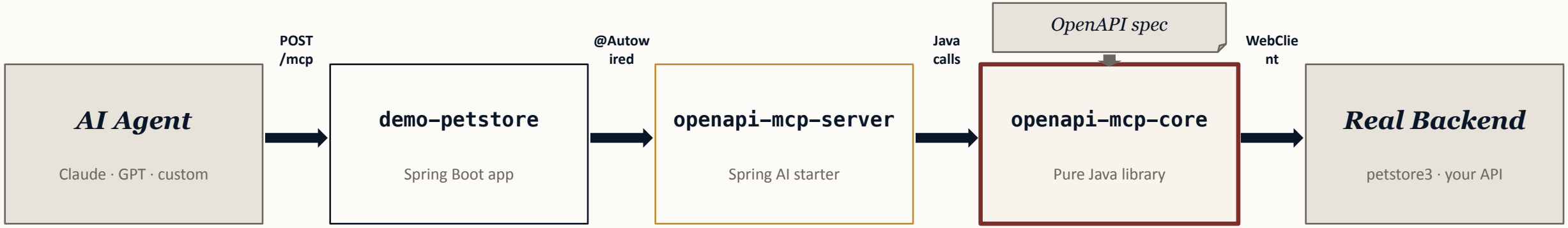
Path templates, query encoding, header rules, multipart bodies — each parameter location has its own little protocol.

06

Auth + errors at runtime

The spec describes contracts. The runtime needs the plumbing. That's where libraries live, not scripts.

The system, on one page.



`openapi-mcp-core`

Pure Java library

- Parses the OpenAPI spec
- Emits a List<GeneratedTool>
- Binds args · applies auth · calls backend

`openapi-mcp-server`

Spring Boot starter

- Auto-configures the generator at boot
- Wraps tools as Spring AI ToolCallbacks
- Publishes them on /mcp (Streamable HTTP)

`demo-petstore`

Reference application

- Spring Boot app consuming the starter
- Bundles the petstore3 OpenAPI spec
- Web UI for live spec editing

One MCP call, end to end.

Wire on the left. Server-side execution on the right.

ON THE WIRE

```
$ curl -X POST localhost:8090/mcp \
    -H 'Content-Type: application/json' \
    -d '{"method":"tools/call",
         "params":{
           "name":"petstore_getPetById",
           "arguments":{"petId":10}}}'
```

↓ 200 OK · ~80 ms ↓

```
data:{"jsonrpc":"2.0","id":3,"result":{
  "content":[{"type":"text",
    "text":"{ id:10, name:\"rufus\"}"}],
  "isError":false}}
```

INSIDE THE SERVER

REQUEST PROCESSING

The MCP server

looks up the right tool by name

The framework

parses your tool arguments

Argument binding

places each input where the spec said it goes

Auth layer

adds the right credentials for this call

BACKEND CALL

The HTTP client

calls petstore3.swagger.io/api/v3/pet/10

RESPONSE PROCESSING

Response mapping

passes 2xx through, translates failures to MCP error codes

Now, the demo.

01
WATCH FOR
boot
spec is parsed, tools register at /mcp

02
WATCH FOR
call
real petstore returns a real pet

03
WATCH FOR
break
watch the structured error contract

Five things that break in production.

01

Pagination

PAIN

Agents ask for everything. Context budgets explode.

FIX

Inject a sane page limit

Capped before the request goes out

02

File uploads

PAIN

Multipart and binary don't round-trip as JSON tool inputs.

FIX

Accept base64 in, send multipart out

Translation handled for you

03

Polymorphic responses

PAIN

oneOf and anyOf confuse agents that want a fixed shape.

FIX

Preserve oneOf in the tool schema

Agent switches on actual shape returned

04

Auth propagation

PAIN

Per-user tokens can't be a static config value.

FIX

Per-call token from your app

Threads the user's session through every call

05

Tool name collisions

PAIN

Two APIs both define operationId 'getStatus'.

FIX

Auto-prefix and de-duplicate

Each spec gets its own namespace

Four things you can swap.

Everything else is opinionated — we picked the boring, working defaults so you don't have to.

01

Authentication

Plug in your auth.

FOR EXAMPLE

Bearer · JWT · mTLS · per-user OAuth

Tokens come from your code, not from a config file.

02

Error mapping

Translate failures into stable codes.

FOR EXAMPLE

422 → SCHEMA_INVALID · 451 → BLOCKED_REGION

Agents branch on codes, not English sentences.

03

Tool naming

Your namespace, your rules.

FOR EXAMPLE

per-tenant prefix · custom slug · safe de-dup

Pick a convention; the generator follows it.

04

Operation filtering

Ship the smallest tool set.

FOR EXAMPLE

skip-deprecated · exclude-by-tag · read-only mode

The agent only sees what you want it to see.

Fail the build if a tool disappears.

One plugin, one report. Diff it in PRs; catch collisions before agents do.

BUILD CONFIG

```
// build.gradle
plugins {
  id 'com.sg.oamcp.openapi-mcp' version '0.1.0'
}

openApiMcp {
  specLocation = '.../openapi.json'
  baseUrl = '.../api/v3'
  toolNamePrefix = 'petstore_'
}
```

GENERATED REPORT · build/openapi-mcp/tools.txt

```
$ ./gradlew generateMcpReport
> Task :generateMcpReport
Wrote MCP tool report (19 tools)

OpenAPI -> MCP tool report
toolCount: 19

PUT  /pet                  -> petstore_updatePet
POST /pet                  -> petstore_addPet
GET  /pet/findByStatus    -> petstore_findPetsByStatus
GET  /pet/{petId}          -> petstore_getPetById
...
```

Four dials you turn first.

01	**Schema refinement**	*Tune the spec descriptions. Agents need verbs, constraints, examples — not human prose. Treat the spec as a product surface.*
02	**Surface area control**	*All 19 endpoints exposed is a starting point, not an end state. Pin the surface — skip-deprecated, exclude-by-tag, read-only mode.*
03	**Capability tagging**	*Destructive tools need server-side permission checks. The client may or may not show a warning. Your server is the only thing that can actually stop a bad call.*
04	**Rate limits & quotas**	*Token bucket in front of the WebClient so an over-eager agent can't blow your downstream SLA. Generator is honest about pressure; runtime isn't.*

Three things to take back to your team.

01

Generate one MCP server this week.

Pick one OpenAPI spec from your portfolio. Use the demo template. Twenty operations turn into twenty tools in an afternoon.

OUTCOME

1 day → 20 tools

02

Add the build-time check to CI.

Fail the pipeline if a known tool disappears or a name collides. This is the cheap insurance that keeps agents from silently regressing.

OUTCOME

~30 lines of CI config

03

Treat tool descriptions as product copy.

Spend the spec writer's time on the descriptions. Regenerate. Measure agent success rate before and after.

OUTCOME

Cheapest measurable lift