Workshop · 60 Minutes

Building Production-Ready MCP Servers

Lessons from an Open Source Template

Most MCP tutorials stop at "hello world." This session goes further.

Abhishek Kumar · Deepak Koul · Red Hat

github.com/redhat-data-and-ai/template-mcp-server

Before / After

Hello World vs Production

Tutorial MCP Server

It Works on My Laptop

- Single transport, stdio only
- No authentication
- Print statements for logging
- No deployment story
- One file, zero tests

Production MCP Server

It Runs in Kubernetes

- Multiple transports: HTTP, SSE, streamable-HTTP
- OAuth 2.0 with token persistence
- Structured JSON logging
- OpenShift manifests + CI/CD
- Modular, tested, documented

Act I

The Gap

Why most MCP servers never leave the developer's machine.

Understanding the problem before solving it

What nobody tells you.

What Production Actually Needs

Six dimensions beyond "it returns a response."

Transport

3 protocols

HTTP, SSE, streamable-HTTP

Security

OAuth

Persistent token storage

Observability

JSON

Structured logging

Configuration

Env

Pydantic-validated

Deployment

K8s

OpenShift manifests

Quality

CI/CD

Tests, linting, hooks

template-mcp-server covers all six

Open source. Apache 2.0.

The Template

A production-grade, open source MCP server template built at Red Hat. FastMCP + FastAPI at the core, with everything you need to go from prototype to production.

Python 3.12+ with UV package manager, Pydantic configuration, structured logging, OAuth integration, and OpenShift deployment manifests.

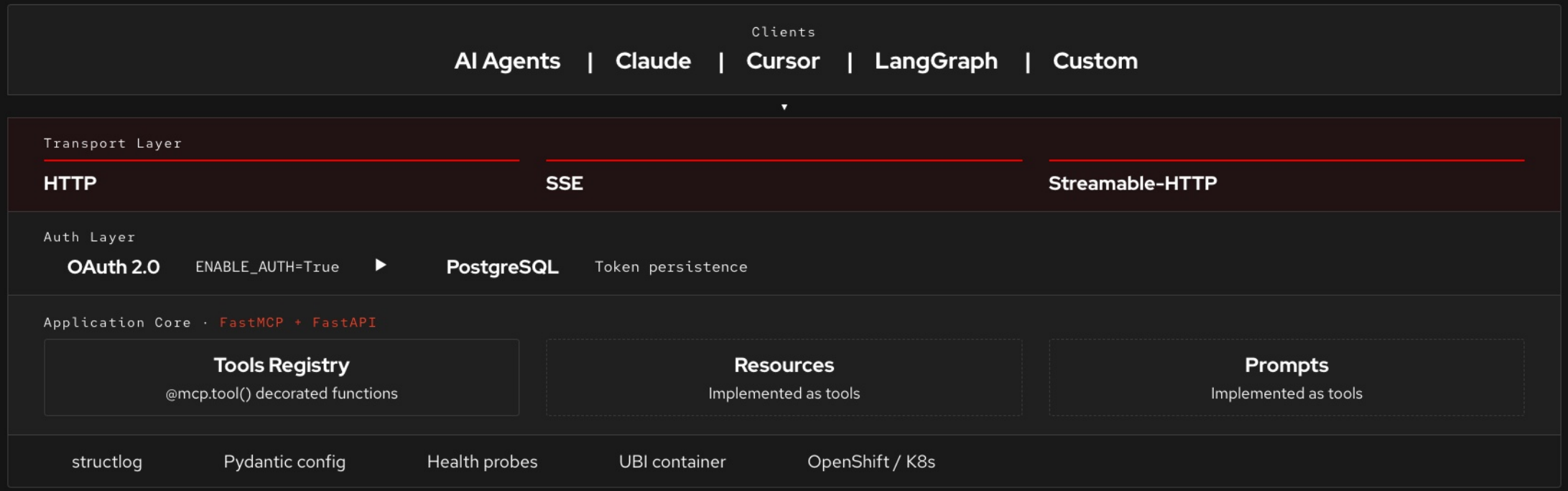
> "Use this template as a starting point for your own MCP servers."
>
> — github.com/redhat-data-and-ai/template-mcp-server

Project Structure

- `template_mcp_server/` main package
- `deployment/openshift/` K8s manifests
- `tests/` unit + integration
- `examples/` FastMCP + LangGraph
- `docs/` comprehensive guides

How it all fits together

Architecture Overview



Act II

Under the Hood

FastMCP, transports, auth, and logging. The four pillars of a production server.

Architecture deep dive

Pillar 01 · Framework

FastMCP + FastAPI

FastMCP provides the MCP protocol layer. FastAPI provides the HTTP backbone. Together, a production server that speaks MCP natively.

```
from fastmcp import FastMCP
from fastapi import FastAPI

mcp = FastMCP("my-server")
app = FastAPI()

@mcp.tool()
def multiply(a: int, b: int) -> int:
    """Multiply two numbers."""
    return a * b
```

Why FastMCP?

- Pythonic decorator API
- Auto schema generation
- Built-in transport support
- Active community

Why FastAPI?

- Async-first design
- OpenAPI docs free
- Middleware ecosystem
- Battle-tested at scale

The best of both worlds

Pillar 02 · Connectivity

Three Transport Protocols

One server, three ways to connect. Switch with a single env var.

HTTP

Standard request-response. Best for simple integrations and load-balanced deployments. Set `MCP_TRANSPORT_PROTOCOL=http`

SSE

Server-Sent Events for real-time streaming. Ideal for long-running tools and progress updates. Set `MCP_TRANSPORT_PROTOCOL=sse`

Streamable-HTTP

The newest MCP transport. Combines HTTP simplicity with streaming capability. Set `MCP_TRANSPORT_PROTOCOL=streamable-http`

> All three transports share the same tool definitions. Your business logic never changes.
>
> — Zero code changes to switch protocols

Pillar 03 · Authentication

OAuth 2.0

Enterprise-grade authentication with persistent token storage in PostgreSQL. Toggle with a single environment variable.

```
# Enable OAuth
ENABLE_AUTH=True

# PostgreSQL for token persistence
DATABASE_URL=postgresql://...

# Tokens survive server restarts
# Sessions persist across deployments
```

> "In production, `ENABLE_AUTH=False` is not an option. The template makes auth opt-in during development and mandatory in production."
>
> — Design decision

Auth Flow

1. Client requests OAuth token
2. Server validates credentials
3. Token stored in PostgreSQL
4. Subsequent requests use token
5. Token refresh handled automatically

Why PostgreSQL?

Tokens survive pod restarts. Shared across replicas. Audit-friendly. Production-proven.

Pillar 04 · Logging

Structured Logging

`print()` does not scale. structlog gives you machine-parseable, human-readable logs from day one.

```
# What you write
logger.info("tool_executed",
    tool="multiply", result=42,
    duration_ms=1.2)

# What you get (JSON in production)
{"event": "tool_executed",
 "tool": "multiply",
 "result": 42,
 "duration_ms": 1.2,
 "level": "info"}
```

structlog benefits

- Context binding
- JSON + console output
- Request correlation
- Zero config in template

Act III

Going to Production

Configuration, containers, Kubernetes, and CI/CD.

Pydantic-powered settings

Environment-Based Config

Configuration Flow

01
Define
Pydantic model with defaults

02
Load
Env vars, .env, ConfigMaps

03
Validate
Type check + constraints

04
Inject
Available app-wide

05
Override
K8s secrets per env

Key Environment Variables

MCP_HOST
localhost

MCP_PORT
5001

ENABLE_AUTH
False

LOG_LEVEL
INFO

No magic. No hidden config. Everything explicit.

Container-first design

UBI Base Image

Red Hat Universal Base Image. Production-hardened, CVE-scanned, enterprise-supported.

```
FROM registry.access.redhat.com/ubi9/python-312
COPY . /app
WORKDIR /app
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5001
CMD ["python", "-m", "template_mcp_server"]
```

Why UBI?

- RHEL-compatible
- CVE-scanned daily
- Free to use
- Minimal attack surface

Build and run

`make container-build`
`make container-run`

Podman or Docker. Same Dockerfile.

OpenShift-ready from day one

Deployment Manifests

Deploy to OpenShift / Kubernetes

01
Build
Container image from UBI base

02
Push
Push to Quay.io or internal registry

03
Configure
Kustomize overlays for dev / staging / prod

04
Deploy
Apply manifests with oc apply or kubectl

05
Monitor
Liveness + readiness probes, health endpoints

Included Manifests

Deployment
Pod spec, replicas, resources

Service
Internal networking and port mapping

Route
External access with TLS termination

Secrets
OAuth creds, DB connection strings

deployment/openshift/ has everything you need

The hard truth

"The last 20% of production readiness takes 80% of the effort."

That is exactly the 20% this template gives you for free.

Built into the template.

CI/CD and Quality

GitHub Actions workflows included out of the box.

Testing

pytest

Unit + integration + container tests

Linting & Formatting

ruff

Fast all-in-one linter and formatter

Pre-commit

hooks

Catch issues before they reach CI

Coverage

codecov

Track and report test coverage

Security

scan

Automated vulnerability scanning

Every PR goes through the full pipeline

What we chose and why

Key Design Decisions

Alternatives we considered

The Road Not Taken

- TypeScript SDK (locked to Node ecosystem)
- Custom auth middleware (reinventing the wheel)
- File-based logging (doesn't scale in K8s)
- Docker Compose only (no K8s story)
- Resources + Prompts registries (client compat issues)

What we chose

The Decisions That Stuck

- Python + FastMCP (widest AI ecosystem reach)
- OAuth 2.0 standard (enterprise-compatible)
- structlog JSON (cloud-native observability)
- OpenShift manifests (production Kubernetes)
- Tools-first architecture (universal client support)

Every decision has a tradeoff. Know yours.

Act IV

Hands-On Lab

Clone. Build a tool. Run. Test with Postman. Connect an agent.

Open your terminal and follow along

Hands-on · Clone and setup

Clone the Template

```
# Option A: Use GitHub template button
# github.com/redhat-data-and-ai/template-mcp-server
# Click "Use this template" → "Create new repo"

# Option B: Clone directly
git clone https://github.com/redhat-data-and-ai\
/template-mcp-server.git
cd template-mcp-server

# Install dependencies (uses UV)
make setup

# Verify installation
make test
```

Prerequisites

Python 3.12+
Git
Make
UV (auto-installed by make setup)

What you get

Full project with example tools, tests, CI/CD, Dockerfile, and OpenShift manifests ready to customize.

make setup handles everything

Hands-on · Write your tool

Add a Custom Tool

Open `template_mcp_server/tools/` and create a new file:

```python
# tools/weather.py
from fastmcp import FastMCP

mcp = FastMCP("weather")

@mcp.tool()
def get_weather(city: str) -> dict:
    """Get current weather for a city."""
    return {
        "city": city,
        "temp": "28°C",
        "condition": "Sunny"
    }
```

One decorator. Full MCP compliance.

That's it

The `@mcp.tool()` decorator does the heavy lifting:

- Auto-generates JSON schema
- Validates input types
- Handles serialization
- Registers with MCP protocol

> Type hints become the tool's schema. Docstrings become the tool's description for the AI agent.
>
> — Python types = MCP schema

Hands-on · Run locally

Start the Server

```
# Start the MCP server
make run
# http://localhost:5001

# Check health
curl http://localhost:5001/health

# List available tools
curl http://localhost:5001/mcp/tools

# Call your tool
curl -X POST http://localhost:5001/mcp \
  -H "Content-Type: application/json" \
  -d '{"method": "tools/call",
       "params": {"name": "get_weather",
       "arguments": {"city": "Mumbai"}}}'
```

Expected output

```
{"city": "Mumbai",
 "temp": "28°C",
 "condition": "Sunny",
 "humidity": "65%"}
```

Transport switch

`MCP_TRANSPORT_PROTOCOL=sse make run`

Same tools, different transport. Try all three.

Hands-on · Verify your server

Test with Postman or MCP Inspector

Postman

1. Create new POST request

2. URL: `http://localhost:5001/mcp`

3. Body (JSON):

```
{"jsonrpc": "2.0", "id": 1,
 "method": "tools/call",
 "params": {
   "name": "get_weather",
   "arguments": {"city": "Mumbai"}
 }
}
```

MCP Inspector

```
npx @modelcontextprotocol/inspector \
  http://localhost:5001/mcp
```

Visual UI to browse tools, test inputs, and inspect responses in real time.

OpenAPI docs

FastAPI auto-generates interactive docs at `/docs` — browse and test all endpoints from the browser.

Hands-on · Wire it up

Connect to an AI Agent

Claude Desktop / VS Code

```
// claude_desktop_config.json
{
  "mcpServers": {
    "my-server": {
      "url": "http://localhost:5001/mcp"
    }
  }
}
```

LangGraph / LangChain

```
from langchain_mcp_adapters import MCPClient
client = MCPClient("http://localhost:5001/mcp")
tools = client.get_tools()
```

Try it

Ask your AI agent: *"What's the weather in Mumbai?"*
The agent discovers your tool, calls it, and returns the response.

Any MCP client works

Claude, Cursor, Windsurf, Continue, Cline, LangGraph, custom agents.

So you don't have to

Mistakes We Avoided

Coupling tool logic to transport layer. Tools should be transport-agnostic.	Architecture
Storing tokens in memory. Pod restarts kill all sessions. Use PostgreSQL.	Security
Using Resources and Prompts registries. Many clients only support Tools.	Compatibility
Hardcoded config values. Everything must come from environment variables.	Operations
Skipping SSL/TLS in staging. Certificate issues always surface in production.	Security

Learn from our mistakes, not yours

Lessons

Why this matters.

Benefits to the MCP Ecosystem

Open source means everyone wins.

Faster Adoption

Developers skip months of boilerplate. Clone, customize, deploy. Production-ready MCP servers in hours, not weeks.

Enterprise Confidence

OAuth, structured logging, and Kubernetes deployment remove the barriers that keep MCP out of enterprise environments.

Community Standard

An open, opinionated template establishes patterns that the community can build on, fork, and improve together.

Apache 2.0 licensed · Free to use and modify

Your move

What will you build with your first production MCP server?

Database connectors. API orchestrators. Business rule engines.
The protocol is the platform.

The template handles the infrastructure. You handle the innovation.

Everything you need

Get Started Today

github.com/redhat-data-and-ai/template-mcp-server	Source
aitemplates.io/templates/mcp-server	Docs
Architecture docs, deployment guides, 5-minute tutorials	Guides
FastMCP + LangGraph integration examples	Examples

Quick start

```
git clone https://github.com/\
redhat-data-and-ai/template-mcp-server
cd template-mcp-server
make setup
make run
# Live at http://localhost:5001
```

Open source. Apache 2.0. Free to use.

What to remember

"Don't build MCP infrastructure. Build MCP tools. Let the template handle the rest."

Your value is in the tools you create, not the plumbing around them.

— Abhishek Kumar & Deepak Koul · MCP Dev Summit Mumbai 2026

Focus on what matters

Thank you

Questions?

Let's build production-ready MCP servers together.

github.com/redhat-data-and-ai/template-mcp-server

aitemplates.io/templates/mcp-server

Abhishek Kumar · Deepak Koul · Red Hat