Build Your First MCP Server

A Developer's Guide to Wrapping Existing APIs for AI Agents to Use

MATTHEW GROFF

Build Your First MCP Server

Matthew Groff

Table of Contents

1 About the Author

Matthew Groff

Matthew Groff is a Principal AI Engineer and AI Capability Lead at Umbrage, part of Bain & Company, based in Orlando, FL. He specializes in AI Product Engineering, web application development, and leading technical teams.

With extensive experience in both AI systems and modern web technologies, Matthew brings practical, real-world expertise to help developers bridge the gap between traditional APIs and AI-powered applications. He shares insights about artificial intelligence and web development through his blog at groff.dev and on LinkedIn.

You can find Matthew online:

- Website: https://groff.dev
- GitHub: https://github.com/mattlgroff
- LinkedIn: https://www.linkedin.com/in/mattgroff/
- YouTube: https://www.youtube.com/@MattTheAIWebDev

1.1 Professional Disclaimer

All content, opinions, and consulting services offered here are Matthew's personal work and views. They are not affiliated with, endorsed by, or representative of Umbrage or Bain & Company. His independent consulting through Groff Dev LLC operates separately from his employment responsibilities.

2 What is an MCP Server?

Imagine you have a toolbox filled with specialized tools. Each tool has a clear label describing when you'd want to use it and how it works. Now imagine that instead of just one toolbox, you could have multiple toolboxes, each containing tools for different domains - one for travel planning, another for file management, and yet another for database operations.

This is exactly what MCP (Model Context Protocol) servers provide for AI agents. An MCP server is essentially a specialized toolbox that exposes a collection of tools, resources, and prompts through a standardized interface that AI agents can discover and use.

2.1 The Toolbox Analogy

Think of an MCP server as a digital toolbox with three types of items:

2.1.1 Tools - The Action Items

Tools are like the actual implements in your toolbox - each one performs a specific action when called upon. Just as you might have a hammer for driving nails or a screwdriver for turning screws, MCP tools perform specific operations like:

- `searchFlights` - Find available flights between cities
- `sendEmail` - Send an email message
- `createCalendarEvent` - Add an event to a calendar
- `queryDatabase` - Execute a database query

Each tool comes with a clear label (its schema) that describes:

- What it does
- What inputs it needs
- What outputs it provides
- When you'd want to use it

2.1.2 Resources - The Reference Materials

Resources are like the instruction manuals, blueprints, and reference materials you might keep alongside your tools. They provide context and information that AI agents need to make informed decisions:

- Calendar data to check availability
- Travel documents for important information
- Previous project files for reference
- Weather data for planning

2.1.3 Prompts - The Templates

Prompts are like pre-written templates or checklists that guide how to use the tools effectively. They provide structured workflows for common tasks:

- "Plan a vacation" - A template that guides through the entire vacation planning process
- "Analyze sales data" - A structured approach to examining business metrics
- "Debug application" - A systematic troubleshooting workflow

2.2 How AI Agents Use MCP Servers

AI agents act as intelligent coordinators that can:

1. **Discover available toolboxes** - Find and connect to MCP servers
2. **Examine the tools** - Read the labels and understand what each tool does
3. **Plan their approach** - Decide which tools to use and in what order
4. **Execute the plan** - Call the appropriate tools with the right parameters
5. **Coordinate across toolboxes** - Use tools from multiple servers together

The agent's main responsibilities become:

- **Intent recognition** - Understanding what the user wants to accomplish
- **Planning** - Figuring out which tools to use and how
- **Execution** - Making the actual tool calls
- **Coordination** - Orchestrating multiple tools to achieve complex results

2.3 The Power of Reusability

One of the most powerful aspects of MCP servers is their reusability. Once you build a toolbox (MCP server), any number of AI agents can use it. This means:

- A travel planning toolbox can be used by personal assistants, business travel apps, and vacation planning services
- A database toolbox can serve web applications, analytics dashboards, and reporting systems
- A file management toolbox can support document editors, backup systems, and content management platforms

2.4 Wrapping Existing APIs

In this ebook, we'll focus primarily on creating MCP servers that wrap existing APIs into tools that AI agents can call. This approach allows you to leverage existing services and APIs that already exist without having to re-implement them.

For example, instead of an AI agent needing to understand the specific quirks of the GitHub API, Slack API, and Google Calendar API, you can create MCP servers that expose clean, standardized tools like:

- `createGitHubIssue`
- `sendSlackMessage`
- `scheduleCalendarEvent`

The AI agent calls these tools with the appropriate parameters, and your MCP server handles all the API-specific details.

2.5 A Simple Mental Model

Here's a simple way to think about MCP servers:

MCP Server = Specialized Toolbox

- Contains tools (actions the AI can take)
- Contains resources (information the AI can access)
- Contains prompts (templates for common workflows)
- Has a standardized interface (so any AI agent can use it)
- Is reusable (multiple AI agents can share the same toolbox)

AI Agent = Intelligent Coordinator

- Discovers available toolboxes
- Understands user intent
- Plans which tools to use
- Executes tool calls
- Coordinates multiple toolboxes

In the next chapter, we'll start building your first MCP server by setting up the development environment with Bun, a fast JavaScript/TypeScript runtime that makes building MCP servers straightforward and efficient.

3 Installing Bun

You can read more about Bun on their website. If these commands have become outdated, check their website for the latest instructions.

3.1 Installing Bun on Windows

Open PowerShell and run the following command:

```
powershell -c "irm bun.sh/install.ps1 | iex"
```

3.2 Installing Bun on macOS or Linux

Type the following in your terminal:

```
curl -fsSL https://bun.sh/install | bash
```

3.3 Verify Bun Installation

That's it! Bun is now installed. You can test it by running:

```
bun --version
```

If you see the Bun version number, you're good to go.

4 Setting Up WeatherAPI.com

Before we can build our first MCP server, we need access to a real API that provides useful data. For our example, we'll use WeatherAPI.com - a free weather service that provides a simple API secured by an API key.

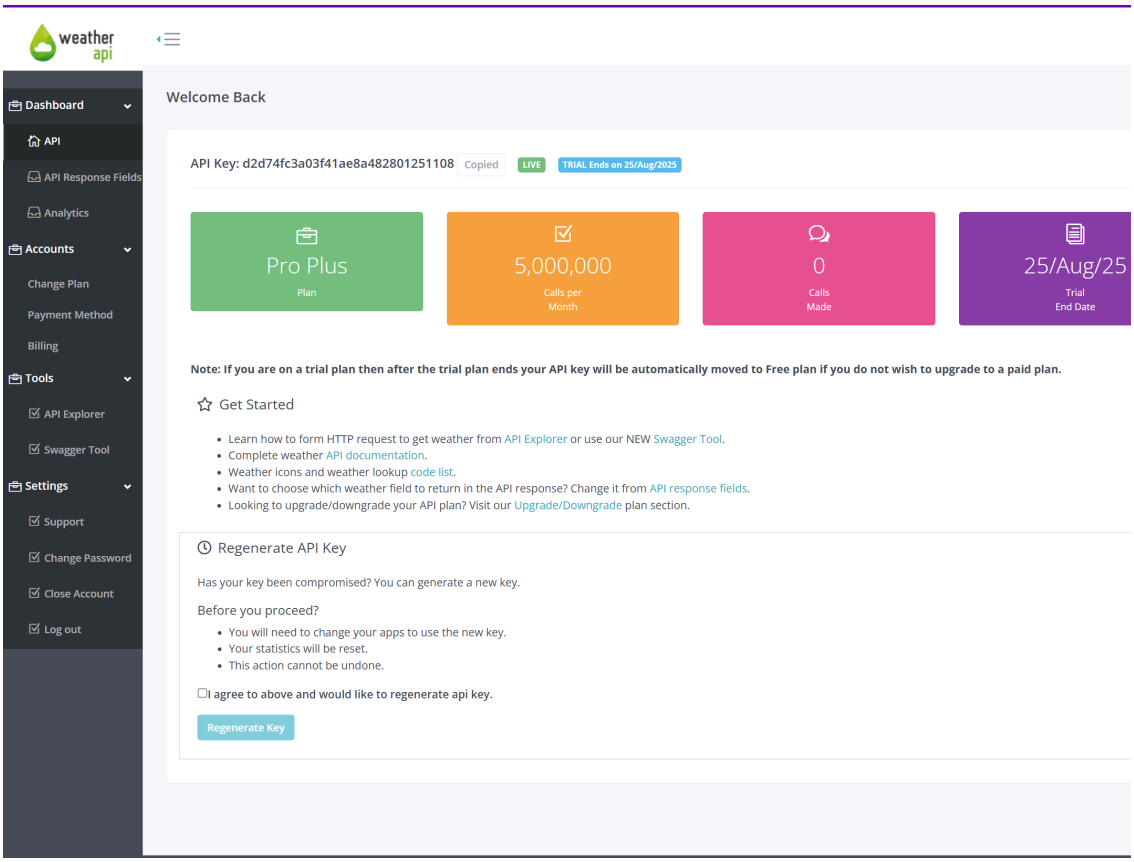
Disclaimer: I am not affiliated with WeatherAPI.com and this is not an endorsement. We're simply using them as an easy example of an API secured by an API key that happens to be free, making it ideal for learning MCP server development.

4.1 Creating Your Account

1. **Visit WeatherAPI.com** - Go to https://www.weatherapi.com/
2. **Sign up for free** - Click the "Sign Up" button and create your account with:
   - Your email address
   - A secure password
   - Basic account information
3. **Verify your email** - Check your inbox for a verification email from WeatherAPI.com and click the verification link. This step is required before you can access your API key.
4. **Log in to your account** - Once verified, log in to access your dashboard

4.2 Getting Your API Key

After logging in and verifying your email, you'll need to locate your API key:



WeatherAPI Dashboard

Your API key is displayed prominently on your dashboard. This key is what allows your MCP server to authenticate with WeatherAPI.com and make requests.

Important Security Note: The API key shown in the screenshot above has been rotated and is no longer active. You'll need to use your own API key from your account.

4.3 API Endpoints We'll Use

For our MCP server, we'll focus on these key endpoints:

4.3.1 Current Weather

```
GET http://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&q=LOCATION
```

4.3.2 Weather Forecast

```
GET http://api.weatherapi.com/v1/forecast.json?key=YOUR_API_KEY&q=LOCATION&days=3
```

4.3.3 Location Search

```
GET http://api.weatherapi.com/v1/search.json?key=YOUR_API_KEY&q=LOCATION
```

4.4 Testing Your API Key

Before we start building our MCP server, let's verify your API key works:

```
curl "http://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&q=London"
```

Replace `YOUR_API_KEY` with your actual API key. You should receive a JSON response with current weather data for London.

4.5 What's Next

Now that you have your WeatherAPI.com account set up and your API key ready, we're prepared to build our first MCP server. In the next chapter, we'll create a weather MCP server that wraps these API endpoints into tools that AI agents can discover and use.

Our weather MCP server will provide tools like:

- `getCurrentWeather` - Get current conditions for any location
- `getWeatherForecast` - Get multi-day weather predictions
- `searchLocations` - Find and validate location names

This will demonstrate the core concepts of MCP server development while creating something genuinely useful for AI applications.

5 The Three Essential Conversations

Building an MCP server is like creating a helpful assistant that can have three types of conversations with AI agents. Think of it as meeting someone new at a party - there's a natural flow to how these interactions work.

5.1 The Social Dance of MCP

When an AI agent meets your MCP server for the first time, they go through the same social ritual humans do:

5.1.1 1. The Introduction: "How do you do?"

MCP Method: `initialize`

Just like when you meet someone new, the first thing that happens is introductions. The AI agent says "Hello, who are you?" and your server responds with: - Its name and version - What it's capable of doing - Which version of the MCP protocol it speaks

This is the handshake - a polite way of establishing who's who and what's possible.

5.1.2 2. The Interest Check: "What can you do?"

MCP Method: `tools/list`

Once introductions are done, the AI agent gets curious: "That's interesting, what tools do you have in your toolbox?" Your server then shows off its capabilities by listing all available tools with descriptions of what each one does.

It's like someone asking "What's your job?" and you explaining not just your title, but what you actually do day-to-day.

5.1.3 3. The Request: "Can you help me with this?"

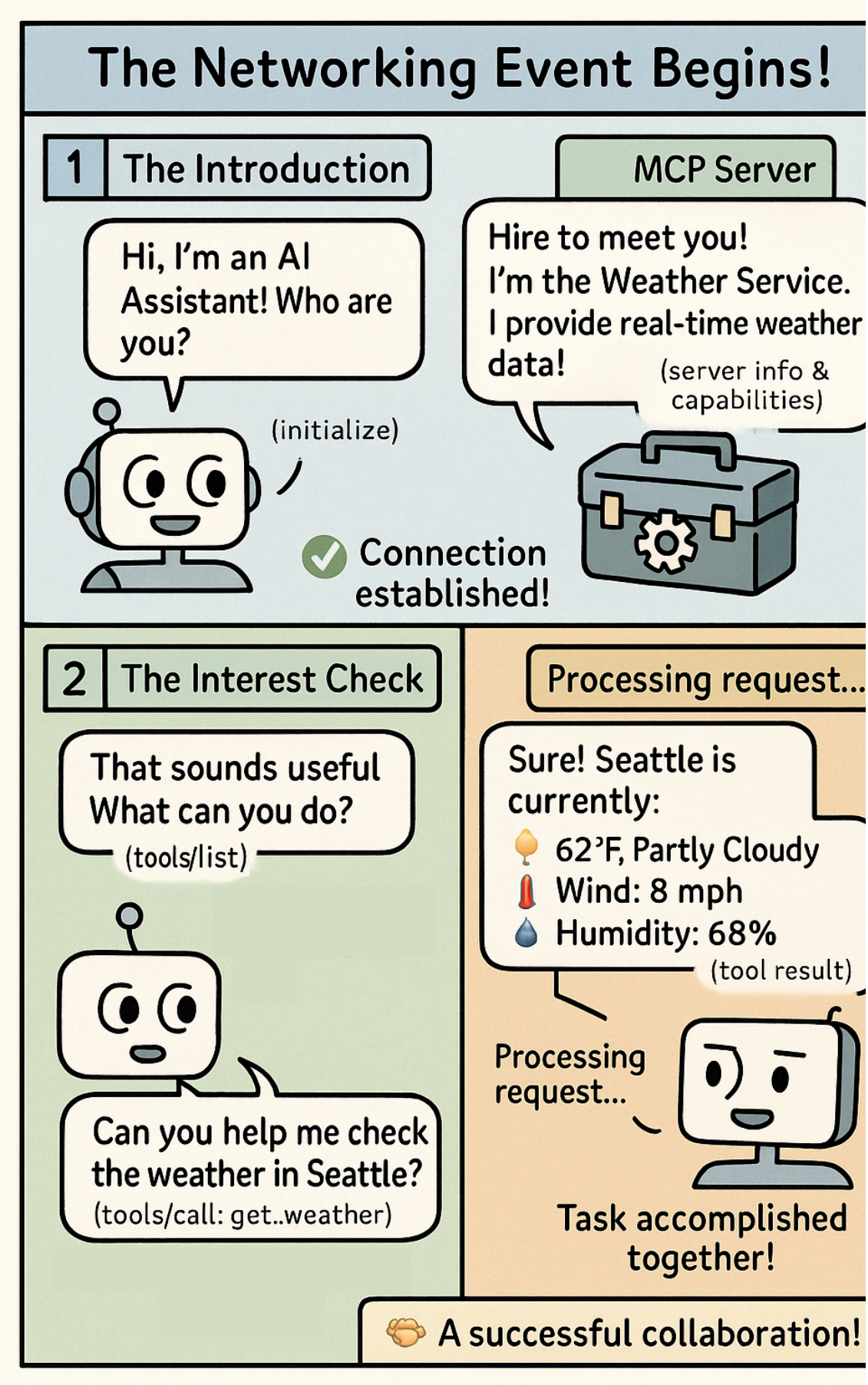
MCP Method: `tools/call`

Finally, the AI agent has a specific task: "I need to know the weather in Tokyo, can you help?" Your server takes the request, uses the appropriate tool, and returns the results.

This is where the real work happens - like someone asking you to actually do something you said you could do.

5.2 The Conversation Flow

Here's how these three conversations work together:



MCP Conversation Flow

5.3 Why These Three Conversations Matter

Every MCP server, no matter how simple or complex, must handle these three conversations. They're the foundation of the Model Context Protocol:

- **Without** `initialize` : The AI agent doesn't know who you are or what you can do
- **Without** `tools/list` : The AI agent can't discover what tools are available
- **Without** `tools/call` : The AI agent can't actually use your tools

5.4 Our Weather Server's Personality

Our weather MCP server will have a clear personality in these conversations:

Name: `weatherapi-mcp` **Specialty**: Weather information **Tools**: Three weather-related tools **Requirement**: Clients must bring their own WeatherAPI.com key

Think of it as a weather expert who's happy to help, but you need to provide your own access credentials to the weather service they use.

5.5 The Technical Foundation

Under the hood, these conversations happen using JSON-RPC 2.0 over HTTP. But don't worry about the technical details yet - think of JSON-RPC as just the "language" that MCP servers and AI agents use to talk to each other.

All three conversations happen through POST requests to a single endpoint: `/mcp`

The AI agent sends a JSON message describing what it wants, and your server responds with a JSON message containing the answer.

5.6 What's Next

In the following chapters, we'll dive deep into each of these three conversations:

- **Chapter 6**: The handshake - how your server introduces itself
- **Chapter 7**: The toolbox reveal - how to present your weather tools
- **Chapter 8**: The actual work - wrapping WeatherAPI.com calls
- **Chapter 9**: Security and errors - handling authentication and problems

By the end, you'll understand exactly how to build a weather MCP server that can have all three essential conversations with any AI agent.

6 The Handshake: Initialize

Imagine you're at a networking event. Someone walks up to you and says, "Hi, I'm Sarah, I'm a graphic designer. I specialize in logo design and brand identity. Nice to meet you!"

That's exactly what the `initialize` method does for your MCP server - it's the "How do you do?" moment where your server introduces itself to an AI agent.

6.1 The AI Agent Approaches

When an AI agent first connects to your weather MCP server, it sends a polite introduction request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2024-11-05",
    "capabilities": {},
    "clientInfo": {
      "name": "some-ai-agent",
      "version": "1.0.0"
    }
  }
}
```

Think of this as the AI agent saying: "Hello! I'm an AI agent, I speak MCP protocol version 2024-11-05, and I'd like to know who you are and what you can do."

6.2 Your Server's Response

Your weather server responds like a friendly professional at that networking event:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2024-11-05",
    "capabilities": {
      "tools": { "listChanged": true }
    },
    "serverInfo": {
      "name": "weatherapi-mcp",
      "version": "1.0.0"
    }
  }
}
```

This translates to: "Nice to meet you! I'm weatherapi-mcp version 1.0.0. I also speak MCP protocol 2024-11-05, and I have tools available that you can discover and use."

6.3 Breaking Down the Handshake

Let's understand each part of this introduction:

6.3.1 Protocol Version

Both sides confirm they speak the same "language" - MCP protocol version 2024-11-05. It's like confirming you both speak English before having a conversation.

6.3.2 Capabilities

Your server announces what it can do. In our case: - `"tools": { "listChanged": true }` means "I have tools, and I can tell you when my tool list changes"

Note: `listChanged` is not important for our Stateless Streamable HTTP Servers in this eBook as our list of tools is static. If you change the tools available based on headers or Role Based Access Controls you would want this to be `true`.

6.3.3 Server Info

This is your server's business card: - **Name**: `weatherapi-mcp` (what to call your server) - **Version**: `1.0.0` (which version of your server this is)

6.4 The Code Implementation

Here's how you implement the handshake in your weather server using Bun's HTTP server:

```
import { serve } from 'bun';

async function handleMCPRequest(request: MCPRequest): Promise<MCPResponse | null> {
  switch (request.method) {
    case 'initialize':
      return {
        jsonrpc: '2.0',
        id: request.id!,
        result: {
          protocolVersion: '2024-11-05',
          capabilities: {
            tools: { listChanged: true },
          },
          serverInfo: {
              name: 'weatherapi-mcp',
              version: '1.0.0',
          }
        },
      };

    // We'll implement tools/list in Chapter 7
    case 'tools/list':
      // TODO: Chapter 7 - Return available weather tools
      return null;

    // We'll implement tools/call in Chapter 8
    case 'tools/call':
      // TODO: Chapter 8 - Execute weather API calls
      return null;

    default:
      return {
        jsonrpc: '2.0',
        id: request.id || 'unknown',
        error: {
          code: -32601,
          message: `Method ${request.method} not found`,
        },
      };
  }
}

// HTTP Server setup
const port = parseInt(process.env.PORT ?? '3000', 10);

serve({
  port,
  async fetch(req) {
    const url = new URL(req.url);

    // Handle CORS preflight requests
    if (req.method === 'OPTIONS') {
      return new Response(null, {
        headers: {
          'Access-Control-Allow-Origin': '*',
          'Access-Control-Allow-Methods': 'POST, GET, OPTIONS',
          'Access-Control-Allow-Headers': '*',
        },
      });
    }

    // Only accept POST requests to /mcp
    if (req.method !== 'POST' || url.pathname !== '/mcp') {
      return new Response('Not found', { status: 404 });
    }

    try {
      const body = await req.json() as MCPRequest;
      const response = await handleMCPRequest(body);
```

```
      if (response) {
        return new Response(JSON.stringify(response), {
          headers: {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*',
          },
        });
      } else {
        return new Response(null, {
          status: 204,
          headers: {
            'Access-Control-Allow-Origin': '*',
          },
        });
      }
    } catch (error) {
      const errorResponse: MCPResponse = {
        jsonrpc: '2.0',
        id: 'unknown',
        error: {
          code: -32700,
          message: 'Parse error',
        },
      };

      return new Response(JSON.stringify(errorResponse), {
        status: 400,
        headers: {
          'Content-Type': 'application/json',
          'Access-Control-Allow-Origin': '*',
        },
      });
    }
  },
});

console.log(`🌤️  Weather MCP Server listening on port ${port}`);
console.log(`🔗 Ready to handle MCP requests at http://localhost:${port}/mcp`);
```

6.5 Why This Matters

The initialize handshake is crucial because:

1. **Version Compatibility**: Both sides confirm they can communicate
2. **Identity Establishment**: Your server gets to introduce itself properly
3. **Trust Building**: Like any good first impression, it sets the tone

6.6 What Happens Next

After this handshake, the AI agent knows: - Your server's name and version - That you both speak the same protocol version

The AI agent will typically follow up by asking "What tools do you have?" - which we'll cover in the next chapter about `tools/list`.

6.7 A Real-World Analogy

Think of `initialize` like the first few seconds when you meet someone:

Bad first impression: "Uh, hi, I'm… um… I do stuff with computers or whatever."

Good first impression: "Hello! I'm Alex, I'm a weather data specialist."

Your MCP server's `initialize` response is its chance to make that good first impression with every AI agent it meets.

6.8 Common Mistakes to Avoid

- **Don't forget the protocol version**: AI agents need to know you speak the same language
- **Don't use generic names**: "weatherapi-mcp" is much better than "my-server"
- **Don't forget the ID**: Always include the request ID in your response

The handshake sets the stage for everything that follows.

7 Showing Your Toolbox: Tools List

After the handshake, the AI agent is curious. It's like someone at that networking event saying, "That's interesting! What exactly do you do? What services do you offer?"

This is where `tools/list` comes in - your chance to open up your toolbox and show off what's inside.

7.1 The AI Agent's Curiosity

The AI agent sends a simple request:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/list"
}
```

This is the AI equivalent of asking: "Can you show me what's in your toolbox?"

7.2 Opening Your Weather Toolbox

Your weather server responds by laying out all its tools like a craftsperson showing their instruments:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "tools": [
      {
        "name": "getCurrentWeather",
        "title": "Get current weather conditions",
        "description": "Get real-time weather for any location - like checking if it's raining right now in Pa
        "inputSchema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "Any location: city name, ZIP code, coordinates, or even an IP address"
            }
          },
          "required": ["location"]
        }
      },
      {
        "name": "getWeatherForecast",
        "title": "Get weather forecast",
        "description": "Get weather predictions for the next 1-3 days - perfect for planning weekend trips",
        "inputSchema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "Any location: city name, ZIP code, coordinates, or even an IP address"
            },
            "days": {
              "type": "integer",
              "description": "How many days ahead (1-3 for free accounts)",
              "minimum": 1,
              "maximum": 3,
              "default": 3
            }
          },
          "required": ["location"]
        }
      },
      {
        "name": "searchLocations",
        "title": "Search for locations",
        "description": "Find the right location name - useful when you're not sure how to spell 'Albuquerque'"
        "inputSchema": {
          "type": "object",
          "properties": {
            "query": {
```

```
              "type": "string",
              "description": "Search for any location - partial names work too!"
            }
          },
          "required": ["query"]
        }
      }
    ]
  }
}
```

7.3 Understanding Each Tool's Label

Just like tools in a real toolbox have labels explaining what they do, each MCP tool has several important pieces of information:

7.3.1 Tool Name

The unique identifier - like the model number on a power drill. The AI agent uses this exact name when it wants to use the tool.

7.3.2 Tool Title

A human-friendly name - like "Cordless Drill" instead of "Model XR-2000-B". This helps AI agents understand what the tool does at a glance.

7.3.3 Tool Description

The detailed explanation - like the description on a tool's packaging that explains when and why you'd use it. Make these descriptions helpful and specific!

7.3.4 Input Schema

The instruction manual - exactly what information the tool needs to work. This is like the specifications that tell you what size drill bits to use or what voltage the tool requires.

7.4 The Toolbox Analogy in Action

Imagine showing someone your actual toolbox:

Tool 1: Hammer - Name: `hammer` - Title: "Claw Hammer" - Description: "For driving nails and removing them - essential for any construction project" - Input Schema: Requires a nail and a surface to hit

Tool 2: Screwdriver - Name: `screwdriver` - Title: "Phillips Head Screwdriver" - Description: "For tightening and loosening Phillips head screws" - Input Schema: Requires a screw and direction (clockwise/counterclockwise)

Our weather tools work the same way, but instead of physical construction, they build weather information.

7.5 The Code Implementation

Here's how you define your weather toolbox and handle the `tools/list` request:

```
// Define your weather tools
const tools = [
  {
    name: 'getCurrentWeather',
    title: 'Get current weather conditions',
    description: 'Get real-time weather for any location - like checking if it\'s raining right now in Paris',
    inputSchema: {
      type: 'object',
      properties: {
        location: {
          type: 'string',
          description: 'Any location: city name, ZIP code, coordinates, or even an IP address'
        }
      },
      required: ['location']
    }
  },
  {
    name: 'getWeatherForecast',
    title: 'Get weather forecast',
    description: 'Get weather predictions for the next 1-3 days - perfect for planning weekend trips',
    inputSchema: {
```

```
      type: 'object',
      properties: {
        location: {
          type: 'string',
          description: 'Any location: city name, ZIP code, coordinates, or even an IP address'
        },
        days: {
          type: 'integer',
          description: 'How many days ahead (1-3 for free accounts)',
          minimum: 1,
          maximum: 3,
          default: 3
        }
      },
      required: ['location']
    }
  },
  {
    name: 'searchLocations',
    title: 'Search for locations',
    description: 'Find the right location name - useful when you\'re not sure how to spell "Albuquerque"',
    inputSchema: {
      type: 'object',
      properties: {
        query: {
          type: 'string',
          description: 'Search for any location - partial names work too!'
        }
      },
      required: ['query']
    }
  }
];

// Updated MCP request handler with tools/list implementation
async function handleMCPRequest(request: MCPRequest): Promise<MCPResponse | null> {
  switch (request.method) {
    case 'initialize':
      return {
        jsonrpc: '2.0',
        id: request.id!,
        result: {
          protocolVersion: '2024-11-05',
          capabilities: {
            tools: { listChanged: true },
          },
          serverInfo,
        },
      };

    case 'tools/list':
      return {
        jsonrpc: '2.0',
        id: request.id!,
        result: { tools },
      };

    // We'll implement tools/call in Chapter 8
    case 'tools/call':
      // TODO: Chapter 8 - Execute weather API calls
      return null;

    default:
      return {
        jsonrpc: '2.0',
        id: request.id || 'unknown',
        error: {
          code: -32601,
          message: `Method ${request.method} not found`,
        },
      };
  }
}
```

7.6 What Makes a Good Tool Description

Bad description: “Gets weather” **Good description**: “Get real-time weather for any location - like checking if it’s raining right now in Paris”

Bad description: “Searches stuff” **Good description**: “Find the right location name - useful when you’re not sure how to spell ‘Albuquerque’”

Good descriptions: - Explain the specific purpose - Give concrete examples - Mention when it’s useful - Use friendly, conversational language

7.7 Input Schema: The Tool’s Requirements

The input schema is like the requirements list for using a tool. For our weather tools:

- **getCurrentWeather**: Needs a location (required)
- **getWeatherForecast**: Needs a location (required) and optionally how many days (1-3)
- **searchLocations**: Needs a search query (required)

The schema tells the AI agent exactly what information to provide, what’s required vs. optional, and what format to use.

7.8 Why This Step Matters

The `tools/list` response is where AI agents decide whether your server is useful for their current task. If you have a tool called `getCurrentWeather` and the AI agent needs current weather data, it will get excited and want to use your tool.

It’s like someone looking through your toolbox and saying, “Perfect! You have exactly the screwdriver I need!”

7.9 Upgrading to Zod for Better Schema Management

Now that you understand how JSON schemas work, let’s talk about a better way to handle them. Writing JSON schemas by hand and then writing separate validation logic is tedious and error-prone. This is where Zod becomes our secret weapon.

7.9.1 The Problem with Manual JSON Schemas

Looking at our current approach, we have to: 1. Write JSON schemas for our tool definitions (so AI agents know what to send) 2. Write separate validation logic (to check what AI agents actually send) 3. Keep both in sync manually (a recipe for bugs)

Here’s what that looks like the hard way:

```
// JSON schema for the AI agent
const weatherToolSchema = {
  type: 'object',
  properties: {
    location: { type: 'string', description: 'City name or coordinates' },
    units: { type: 'string', enum: ['celsius', 'fahrenheit'], default: 'celsius' }
  },
  required: ['location']
};

// Separate validation logic - easy to get out of sync!
function validateWeatherInput(input: any) {
  if (!input.location || typeof input.location !== 'string') {
    throw new Error('Location must be a string');
  }
  if (input.units && !['celsius', 'fahrenheit'].includes(input.units)) {
    throw new Error('Units must be celsius or fahrenheit');
  }
  // More validation code...
}
```

7.9.2 The Zod Solution: Define Once, Use Everywhere

With Zod, you define your schema once and get both validation and JSON schema generation:

```
import { z } from 'zod';

// Define your schema once with Zod
const getCurrentWeatherSchema = z.object({
  location: z.string().describe('Any location: city name, ZIP code, coordinates, or IP address'),
});

const getWeatherForecastSchema = z.object({
  location: z.string().describe('Any location: city name, ZIP code, coordinates, or IP address'),
  days: z.number().int().min(1).max(3).default(3).describe('Number of forecast days (1-3 for free accounts)'),
```

```
});

const searchLocationsSchema = z.object({
  query: z.string().describe('Search query for location names - partial matches work'),
});

// Convert Zod schemas to JSON schemas for MCP
function zodToMCPSchema(schema: z.ZodObject<any>) {
  const jsonSchema = z.toJSONSchema(schema) as any;

  // Fix the required array - remove fields that are optional or have defaults
  if (jsonSchema.required && Array.isArray(jsonSchema.required)) {
    const shape = schema.shape;
    jsonSchema.required = jsonSchema.required.filter((fieldName: string) => {
      const field = shape[fieldName];
      // Remove from required if field is optional or has a default
      return !field.isOptional() && field._def.defaultValue === undefined;
    });

    // If no truly required fields, remove the required array entirely
    if (jsonSchema.required.length === 0) {
      delete jsonSchema.required;
    }
  }

  return jsonSchema;
}

// Your tools array now uses generated schemas
const tools = [
  {
    name: 'getCurrentWeather',
    title: 'Get current weather conditions',
    description: 'Get real-time weather for any location - like checking if it\'s raining right now in Paris',
    inputSchema: zodToMCPSchema(getCurrentWeatherSchema),
  },
  {
    name: 'getWeatherForecast',
    title: 'Get weather forecast',
    description: 'Get weather predictions for the next 1-3 days - perfect for planning weekend trips',
    inputSchema: zodToMCPSchema(getWeatherForecastSchema),
  },
  {
    name: 'searchLocations',
    title: 'Search for locations',
    description: 'Find the right location name - useful when you\'re not sure how to spell "Albuquerque"',
    inputSchema: zodToMCPSchema(searchLocationsSchema),
  }
];
```

7.9.3 Why This Matters for AI Agents

AI agents make mistakes. They might send: - A number instead of a string: `{ location: 12345 }` - Missing required fields: `{ units: "celsius" }` (no location) - Invalid enum values: `{ location: "Paris", units: "kelvin" }`

When Zod catches these errors, it provides detailed feedback that helps the AI agent fix its mistake:

```
{
  "jsonrpc": "2.0",
  "id": 123,
  "error": {
    "code": -32602,
    "message": "Invalid parameters",
    "data": {
      "issues": [
        {
          "path": ["location"],
          "message": "Required field missing"
        },
        {
          "path": ["days"],
          "message": "Number must be greater than or equal to 1"
        }
      ]
    }
  }
}
```

7.9.4 Setting Up Zod in Your Project

First, install Zod as your only dependency:

```
bun add zod
```

7.9.5 Why Zod is Perfect for MCP Servers

1. **Single Source of Truth**: Define your schema once, use it everywhere
2. **Type Safety**: Get TypeScript types automatically from your schemas
3. **Descriptive Errors**: AI agents get helpful feedback when they make mistakes
4. **Zero Runtime Overhead**: Validation only happens when needed
5. **Industry Standard**: Used by Vercel's AI SDK and many other AI tools for the same reasons

This is why Zod is our only dependency - it solves the schema definition and validation problem elegantly, letting us focus on building great weather tools instead of writing validation boilerplate.

In the next chapter, we'll see how these Zod schemas automatically validate incoming tool calls and provide helpful error messages when AI agents send invalid data.

7.10 What Happens Next

After seeing your toolbox, the AI agent will typically pick one of your tools and ask you to use it. That's where `tools/call` comes in - the actual work of using the tools you've advertised.

But first, the AI agent needs to understand how your tools actually work under the hood, which we'll cover in the next chapter about wrapping API calls.

8 Wrapping the API Calls: The Technical Implementation

Now we get to the meat of the operation - how your tools actually work. This is where we stop talking about toolboxes and start writing code that makes real HTTP requests to WeatherAPI.com.

Think of this as the difference between showing someone a hammer and actually knowing how to swing it effectively.

8.1 The Architecture: Three Layers

Our weather MCP server has three distinct layers:

1. **MCP Layer**: Handles JSON-RPC communication with AI agents
2. **Validation Layer**: Uses our Zod schemas from Chapter 7 to ensure inputs are correct
3. **API Layer**: Makes actual HTTP requests to WeatherAPI.com

8.2 The WeatherAPI.com Wrapper Function

Here's the core function that handles all communication with WeatherAPI.com:

```
 const WEATHER_API_BASE = 'https://api.weatherapi.com/v1';

async function callWeatherAPI(endpoint: string, apiKey: string, params: Record<string, any>) {
  // Build the URL with the API key
  const url = new URL(`${WEATHER_API_BASE}${endpoint}`);
  url.searchParams.set('key', apiKey);

  // Add all the parameters
  Object.entries(params).forEach(([key, value]) => {
    if (value !== undefined && value !== null) {
      url.searchParams.set(key, value.toString());
    }
  });

  // Make the HTTP request
  const response = await fetch(url.toString());

  // Handle errors
  if (!response.ok) {
    const errorText = await response.text();
    console.error(`❌ WeatherAPI error: ${response.status} - ${errorText}`);

    let errorBody;
    try {
      errorBody = JSON.parse(errorText);
    } catch {
      errorBody = { message: errorText };
    }

    throw new WeatherAPIError(
      response.status,
      response.statusText,
      errorBody,
      `WeatherAPI request failed: ${errorText}`
    );
  }

  // Return the JSON response
  return response.json();
}

// Custom error class for WeatherAPI errors
class WeatherAPIError extends Error {
  constructor(
    public status: number,
    public statusText: string,
    public body: any,
    message?: string
  ) {
    super(message || `WeatherAPI error (${status}): ${statusText}`);
    this.name = 'WeatherAPIError';
  }
}
```

8.3 Individual API Wrapper Functions

Now we create specific functions for each WeatherAPI.com endpoint:

```
// Get current weather data
async function getCurrentWeatherData(apiKey: string, params: z.infer<typeof getCurrentWeatherSchema>) {
  return await callWeatherAPI('/current.json', apiKey, {
    q: params.location,
    lang: params.language,
  });
}

// Get weather forecast data
async function getWeatherForecastData(apiKey: string, params: z.infer<typeof getWeatherForecastSchema>) {
  return await callWeatherAPI('/forecast.json', apiKey, {
    q: params.location,
    days: params.days,
    lang: params.language,
  });
}

// Search for locations
async function searchLocationsData(apiKey: string, params: z.infer<typeof searchLocationsSchema>) {
  return await callWeatherAPI('/search.json', apiKey, {
    q: params.query,
  });
}
```

These functions are thin wrappers that translate our clean parameter names into the specific parameter names that WeatherAPI.com expects.

8.4 MCP Tool Implementations

Finally, we create the MCP tool functions that tie everything together:

```
async function getCurrentWeather(apiKey: string, input: any) {
  // Validate the input
  const validatedInput = getCurrentWeatherSchema.parse(input);

  // Make the API call
  const result = await getCurrentWeatherData(apiKey, validatedInput);

  // Format the response for the AI agent
  const location = result.location;
  const current = result.current;

  return {
    content: [
      {
        type: 'text',
        text: `Current weather in ${location.name}, ${location.country}: ${current.condition.text}, ${current.
      },
      {
        type: 'text',
        text: JSON.stringify(result, null, 2),
      },
    ],
  };
}

async function getWeatherForecast(apiKey: string, input: any) {
  // Apply defaults and validate
  const inputWithDefaults = {
    days: 3,
    ...input
  };
  const validatedInput = getWeatherForecastSchema.parse(inputWithDefaults);

  // Make the API call
  const result = await getWeatherForecastData(apiKey, validatedInput);

  // Format the response
  const location = result.location;
  const forecast = result.forecast.forecastday;

  let summary = `${validatedInput.days}-day forecast for ${location.name}, ${location.country}:\n`;
  forecast.forEach((day: any) => {
    summary += `${day.date}: ${day.day.condition.text}, High: ${day.day.maxtemp_c}°C, Low: ${day.day.mintemp_c
```

```
  });

  return {
    content: [
      {
        type: 'text',
        text: summary,
      },
      {
        type: 'text',
        text: JSON.stringify(result, null, 2),
      },
    ],
  };
}

async function searchLocations(apiKey: string, input: any) {
  // Validate the input
  const validatedInput = searchLocationsSchema.parse(input);

  // Make the API call
  const result = await searchLocationsData(apiKey, validatedInput);

  // Format the response
  let summary = `Found ${result.length} locations matching "${validatedInput.query}":\n`;
  result.forEach((location: any) => {
    summary += `${location.name}, ${location.region}, ${location.country} (${location.lat}, ${location.lon})\n
  });

  return {
    content: [
      {
        type: 'text',
        text: summary,
      },
      {
        type: 'text',
        text: JSON.stringify(result, null, 2),
      },
    ],
  };
}
```

8.5 The Tool Registry

We create a registry that maps tool names to their implementation functions:

```
const allTools = {
  'getCurrentWeather': getCurrentWeather,
  'getWeatherForecast': getWeatherForecast,
  'searchLocations': searchLocations,
};
```

This makes it easy to look up and call the right function when an AI agent requests a specific tool.

8.6 Error Handling Strategy

Our error handling follows a clear pattern:

1. **Validation Errors**: Zod catches bad inputs and throws descriptive errors
2. **API Errors**: WeatherAPI.com errors are caught and wrapped in our custom error class
3. **Network Errors**: Fetch failures are caught and handled gracefully
4. **Unknown Errors**: Everything else gets a generic error message

8.7 Response Format

Every tool returns data in the MCP content format:

```
{
  content: [
    {
      type: 'text',
      text: 'Human-readable summary'
    },
    {
      type: 'text',
      text: 'Full JSON data for programmatic use'
    }
  ]
}
```

This gives AI agents both a summary they can understand and raw data they can process.

8.8 Testing Your API Wrappers

You can test these functions directly:

```
// Test current weather
const weather = await getCurrentWeather('your-api-key', { location: 'London' });
console.log(weather);

// Test forecast
const forecast = await getWeatherForecast('your-api-key', { location: 'Tokyo', days: 2 });
console.log(forecast);

// Test location search
const locations = await searchLocations('your-api-key', { query: 'New York' });
console.log(locations);
```

8.9 What's Next

Now that we have solid API wrappers, we need to handle the security aspect - checking for API keys in the Authorization header and returning proper JSON-RPC errors when things go wrong. That's what we'll cover in the next chapter.

The technical implementation is the foundation that makes everything else possible. Without reliable API wrappers, your MCP server would just be an empty toolbox with broken tools inside.

9 Authentication and Error Handling: The Security Layer

Now we need to add the security layer to our MCP server. This is like having a bouncer at a club - they check IDs and handle problems professionally.

Our server needs to: 1. Check for WeatherAPI.com keys in the Authorization header 2. Return proper JSON-RPC error codes when things go wrong 3. Handle different types of errors gracefully

9.1 The Authorization Header Pattern

Instead of storing API keys in environment variables, we require clients to pass their WeatherAPI.com key through the `Authorization` header using the Bearer token pattern:

```
Authorization: Bearer your-weatherapi-key-here
```

This approach has several advantages: - **Multi-client support**: Different users can use their own keys - **No server-side secrets**: We don't store any API keys - **Standard HTTP pattern**: Uses the widely-adopted Bearer token format - **Flexible deployment**: Same server works for everyone

9.2 Extracting the API Key

Here's how we extract the API key from the request headers:

```
function extractAPIKey(request: Request): string | undefined {
  const authHeader = request.headers.get('Authorization');

  if (!authHeader) {
    console.log('❌ No Authorization header found');
    return undefined;
  }

  if (!authHeader.startsWith('Bearer ')) {
    console.log('❌ Authorization header does not start with "Bearer "');
    return undefined;
  }

  const apiKey = authHeader.slice(7); // Remove "Bearer " prefix

  if (!apiKey || apiKey.trim().length === 0) {
    console.log('❌ Empty API key in Authorization header');
    return undefined;
  }

  console.log('✅ Valid API key found in Authorization header');
  return apiKey.trim();
}
```

9.3 JSON-RPC Error Codes

JSON-RPC has its own set of error codes, similar to HTTP status codes but specific to the protocol. Here are the ones we'll use:

```
const JSON_RPC_ERRORS = {
  PARSE_ERROR: -32700,        // Invalid JSON
  INVALID_REQUEST: -32600,    // Invalid request object
  METHOD_NOT_FOUND: -32601,   // Method doesn't exist
  INVALID_PARAMS: -32602,     // Invalid method parameters
  INTERNAL_ERROR: -32603,     // Internal JSON-RPC error

  // Custom error codes (allowed range: -32000 to -32099)
  UNAUTHORIZED: -32001,       // Missing or invalid API key
  FORBIDDEN: -32002,          // API key lacks permissions
  NOT_FOUND: -32003,          // Resource not found
  RATE_LIMITED: -32004,       // Too many requests
};
```

9.4 Error Response Builder

We need a function that creates properly formatted JSON-RPC error responses:

```
function createErrorResponse(
  id: number | string | null,
  code: number,
  message: string,
  data?: any
): any {
  return {
    jsonrpc: '2.0',
    id: id || 'unknown',
    error: {
      code,
      message,
      ...(data && { data })
    }
  };
}
```

9.5 Mapping WeatherAPI Errors to JSON-RPC

WeatherAPI.com returns HTTP status codes, but we need to translate these to JSON-RPC error codes:

```
function mapWeatherAPIErrorToJSONRPC(error: WeatherAPIError): { code: number, message: string, data?: any } {
  switch (error.status) {
    case 400:
      return {
        code: JSON RPC ERRORS.INVALID PARAMS,
        message: 'Invalid request parameters',
        data: {
          status: error.status,
          detail: error.body?.error?.message || error.message
        }
      };

    case 401:
      return {
        code: JSON RPC ERRORS.UNAUTHORIZED,
        message: 'Invalid WeatherAPI.com API key',
        data: {
          status: error.status,
          detail: 'Check your API key and make sure it\'s active'
        }
      };

    case 403:
      return {
        code: JSON_RPC_ERRORS.FORBIDDEN,
        message: 'WeatherAPI.com access denied',
        data: {
          status: error.status,
          detail: 'Your API key may have exceeded its quota or lacks required permissions'
        }
      };

    case 404:
      return {
        code: JSON RPC ERRORS.NOT FOUND,
        message: 'Location not found',
        data: {
          status: error.status,
          detail: 'The specified location could not be found'
        }
      };

    case 429:
      return {
        code: JSON RPC ERRORS.RATE LIMITED,
        message: 'Rate limit exceeded',
        data: {
          status: error.status,
          detail: 'Too many requests to WeatherAPI.com. Please wait before trying again.'
        }
      };

    default:
      return {
        code: JSON_RPC_ERRORS.INTERNAL_ERROR,
```

```
        message: 'WeatherAPI.com service error',
        data: {
          status: error.status,
          detail: error.message
        }
      };
  }
}
```

9.6 Enhanced MCP Request Handler

Now we update our main request handler to include authentication and proper error handling:

```
async function handleMCPRequest(request: any, apiKey?: string): Promise<any> {
  try {
    switch (request.method) {
      case 'initialize':
        return {
          jsonrpc: '2.0',
          id: request.id,
          result: {
            protocolVersion: '2024-11-05',
            capabilities: { tools: { listChanged: true } },
            serverInfo: { name: 'weatherapi-mcp', version: '1.0.0' },
          },
        };

      case 'tools/list':
        return {
          jsonrpc: '2.0',
          id: request.id,
          result: { tools },
        };

      case 'tools/call':
        const { name, arguments: args } = request.params || {};

        // Check if tool exists
        const tool = allTools[name];
        if (!tool) {
          return createErrorResponse(
            request.id,
            JSON_RPC_ERRORS.METHOD_NOT_FOUND,
            `Tool "${name}" not found`,
            { availableTools: Object.keys(allTools) }
          );
        }

        // Check for API key
        if (!apiKey) {
          return createErrorResponse(
            request.id,
            JSON RPC ERRORS.UNAUTHORIZED,
            'WeatherAPI.com API key required',
            {
              hint: 'Include your API key in the Authorization header: "Bearer your-api-key"',
              signup: 'Get a free API key at https://www.weatherapi.com/signup.aspx'
            }
          );
        }

        // Call the tool
        try {
          const result = await tool(apiKey, args);
          return {
            jsonrpc: '2.0',
            id: request.id,
            result,
          };
        } catch (error) {
          console.error(`❌ Tool "${name}" error:`, error);

          if (error instanceof WeatherAPIError) {
            const { code, message, data } = mapWeatherAPIErrorToJSONRPC(error);
            return createErrorResponse(request.id, code, message, data);
          }

          // Handle Zod validation errors
          if (error.name === 'ZodError') {
            return createErrorResponse(
```

```
              request.id,
              JSON_RPC_ERRORS.INVALID_PARAMS,
              'Invalid input parameters',
              {
                validationErrors: error.errors,
                hint: 'Check the tool\'s input schema for required parameters'
              }
            );
          }

          // Generic error fallback
          return createErrorResponse(
            request.id,
            JSON_RPC_ERRORS.INTERNAL_ERROR,
            'Internal server error',
            { detail: error instanceof Error ? error.message : 'Unknown error' }
          );
        }

      default:
        return createErrorResponse(
          request.id,
          JSON_RPC_ERRORS.METHOD_NOT_FOUND,
          `Method "${request.method}" not found`,
          { availableMethods: ['initialize', 'tools/list', 'tools/call'] }
        );
    }
  } catch (error) {
    console.error('❌ Request handler error:', error);
    return createErrorResponse(
      request.id,
      JSON_RPC_ERRORS.INTERNAL_ERROR,
      'Request processing failed',
      { detail: error instanceof Error ? error.message : 'Unknown error' }
    );
  }
}
```

9.7 Complete HTTP Server with Security

Here's the complete server implementation with authentication and error handling:

```
import { serve } from 'bun';

const port = parseInt(process.env.PORT ?? '3000', 10);

serve({
  port,
  async fetch(req) {
    const url = new URL(req.url);

    // Handle CORS preflight requests
    if (req.method === 'OPTIONS') {
      return new Response(null, {
        headers: {
          'Access-Control-Allow-Origin': '*',
          'Access-Control-Allow-Methods': 'POST, OPTIONS',
          'Access-Control-Allow-Headers': '*',
        },
      });
    }

    // Health check endpoint for monitoring
    if (url.pathname === '/healthz') {
      return new Response('OK', {
        status: 200,
        headers: {
          'Content-Type': 'text/plain',
          'Access-Control-Allow-Origin': '*',
        },
      });
    }

    // Only handle POST requests to /mcp
    if (req.method !== 'POST' || url.pathname !== '/mcp') {
      return new Response('Not Found', {
        status: 404,
        headers: { 'Access-Control-Allow-Origin': '*' }
      });
    }
```

```
    try {
      // Extract API key from Authorization header
      const apiKey = extractAPIKey(req);

      // Parse JSON request body
      const body = await req.json();

      // Validate basic JSON-RPC structure
      if (!body.jsonrpc || body.jsonrpc !== '2.0' || !body.method) {
        const errorResponse = createErrorResponse(
          body.id || null,
          JSON_RPC_ERRORS.INVALID_REQUEST,
          'Invalid JSON-RPC request',
          { hint: 'Request must include jsonrpc: "2.0" and method fields' }
        );

        return new Response(JSON.stringify(errorResponse), {
          status: 400,
          headers: {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*',
          },
        });
      }

      // Handle the MCP request
      const response = await handleMCPRequest(body, apiKey);

      return new Response(JSON.stringify(response), {
        headers: {
          'Content-Type': 'application/json',
          'Access-Control-Allow-Origin': '*',
        },
      });

    } catch (error) {
      console.error('❌ Request processing error:', error);

      const errorResponse = createErrorResponse(
        null,
        JSON_RPC_ERRORS.PARSE_ERROR,
        'Invalid JSON in request body',
        { hint: 'Make sure your request body is valid JSON' }
      );

      return new Response(JSON.stringify(errorResponse), {
        status: 400,
        headers: {
          'Content-Type': 'application/json',
          'Access-Control-Allow-Origin': '*',
        },
      });
    }
  },
});

console.log(`🌤️ Weather MCP Server listening on port ${port}`);
```

9.8 Testing Authentication

You can test the authentication with curl:

```
 # This should fail (no API key)
curl -X POST http://localhost:3000/mcp \
  -H "Content-Type: application/json" \
  -d '{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"getCurrentWeather","arguments":{"locatio

# This should work (with API key)
curl -X POST http://localhost:3000/mcp \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer your-weatherapi-key" \
  -d '{"jsonrpc":"2.0","id":1,"method":"tools/call","params":{"name":"getCurrentWeather","arguments":{"locatio
```

9.9 Error Response Examples

Here are examples of the error responses your server will return:

Missing API Key:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32001,
    "message": "WeatherAPI.com API key required",
    "data": {
      "hint": "Include your API key in the Authorization header: \"Bearer your-api-key\"",
      "signup": "Get a free API key at https://www.weatherapi.com/signup.aspx"
    }
  }
}
```

Invalid Location:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32003,
    "message": "Location not found",
    "data": {
      "status": 404,
      "detail": "The specified location could not be found"
    }
  }
}
```

9.10 Security Considerations

Public Internet Exposure: If you deploy this MCP Server to the public internet, anyone who discovers the URL can access it. While they can't use your API keys (they must provide their own), there's nothing preventing them from adding your server to their MCP client or calling it directly with curl.

Additional Security Options: If you want your MCP Server publicly accessible for your AI agents but restricted from others, consider these approaches:

1. **Secret Header Authentication**: Add an additional secret header (like `x-api-key`) and validate it in your code. With this approach, you could store the WeatherAPI key in environment variables (`process.env.WEATHER_API_KEY`) and authenticate using your secret header instead.
2. **Network-Level Security**: Deploy within a corporate VPN or private network where only authorized users have access. This moves security from the application layer to the network layer.
3. **Public API Strategy**: If you're building a service like WeatherAPI.com, you might actually want public access and encourage users to bring their own API keys.

Choose the approach that matches your use case: - **Public service**: Allow open access with user-provided API keys - **Private company**: Use secret headers or network restrictions
- **Enterprise**: Rely on corporate network security

The security model depends entirely on your intended audience and deployment environment.

10 What We've Accomplished

Our weather MCP server now has: - ✅ Secure API key handling through Authorization headers - ✅ Proper JSON-RPC error codes and messages - ✅ Helpful error details and hints for debugging - ✅ Graceful handling of all error types - ✅ CORS support for web clients - ✅ Health check endpoint for monitoring

The server is now production-ready and can handle real-world usage with proper security and error handling. AI agents will get clear, actionable error messages when something goes wrong, making debugging much easier.

Below is an example Dockerfile you can use to deploy your Bun-based MCP Server. It assumes you named your server file `simple-mcp-server.ts`.

```
 # Use the official Bun image (1.2.2 is the latest stable version as of the writing of this eBook, but you
can increment this to whatever `bun --version` you have)
FROM oven/bun:1.2.2

# Set working directory
WORKDIR /app

# Copy package.json and bun.lockb first for better caching
COPY package.json bun.lockb* ./

# Install dependencies
RUN bun install --frozen-lockfile

# Copy source code
COPY . .

# Expose the port
EXPOSE 3000

# Set default environment variables
ENV PORT=3000
ENV NODE_ENV=production

# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:${PORT}/healthz || exit 1

# Run the application
CMD ["bun", "run", "simple-mcp-server.ts"]
```

You can find the full source code for this Weather API example MCP Server here: https://github.com/mattlgroff/weather-api-mcp-server

11 Setting Up Petfinder API

Now that we've mastered the basics of MCP server development with WeatherAPI, let's explore a more complex authentication pattern by building a Petfinder MCP server. Petfinder uses OAuth 2.0 authentication instead of simple API keys, which will teach us how to handle token-based authentication in MCP servers.

Disclaimer: I am not affiliated with Petfinder and this is not an endorsement. We're using them as an example of an OAuth 2.0 secured API that provides valuable data about adoptable pets.

11.1 Creating Your Petfinder Account

1. **Visit Petfinder Developers** - Go to https://www.petfinder.com/developers/
2. **Sign up for a developer account** - Click "Get an API Key" and create your account with:
   - Your email address
   - A secure password
   - Basic developer information
3. **Create a new application** - After logging in, you'll need to create an application to get your credentials:
   - Application name (e.g., "My MCP Server")
   - Application URL (can be a placeholder like http://localhost)
   - Description of your application

11.2 Getting Your OAuth Credentials

Unlike WeatherAPI's simple API key, Petfinder provides two pieces of information:

- **Client ID** - A public identifier for your application
- **Client Secret** - A private key that must be kept secure

These credentials work together in an OAuth 2.0 flow to obtain temporary access tokens.

Important Security Note: Never commit your Client Secret to version control or share it publicly. It should be treated as securely as a password.

11.3 Understanding OAuth 2.0 vs API Keys

The key difference between Petfinder and WeatherAPI is the authentication method:

11.3.1 WeatherAPI (Simple API Key)

```
GET http://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&q=London
```

11.3.2 Petfinder (OAuth 2.0 Token)

```
 # First: Exchange credentials for token
POST https://api.petfinder.com/v2/oauth2/token
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=YOUR_CLIENT_ID&client_secret=YOUR_CLIENT_SECRET

# Then: Use token in API calls
GET https://api.petfinder.com/v2/animals
Authorization: Bearer YOUR_ACCESS_TOKEN
```

11.4 API Endpoints We'll Use

Our Petfinder MCP server will focus on these endpoints:

11.4.1 Search Animals

```
GET https://api.petfinder.com/v2/animals?type=dog&location=90210
```

11.4.2 Get Animal Details

```
GET https://api.petfinder.com/v2/animals/{id}
```

11.4.3 Search Organizations

```
GET https://api.petfinder.com/v2/organizations?location=90210
```

11.5 Testing Your Credentials

Let's verify your OAuth credentials work by manually obtaining a token:

```
curl -X POST https://api.petfinder.com/v2/oauth2/token \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "grant_type=client_credentials&client_id=YOUR_CLIENT_ID&client_secret=YOUR_CLIENT_SECRET"
```

You should receive a response like:

```
{
  "token type": "Bearer",
  "expires in": 3600,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOi..."
}
```

11.6 What Makes This Different

The OAuth 2.0 pattern introduces several complexities our MCP server must handle:

1. **Token Exchange** - Converting client credentials to access tokens
2. **Token Expiration** - Access tokens expire and must be refreshed
3. **In-Memory Storage** - Caching tokens to avoid unnecessary API calls
4. **Authorization Headers** - Using Bearer tokens instead of query parameters

In the next chapters, we'll implement these patterns to create a robust Petfinder MCP server that handles OAuth authentication seamlessly.

11.7 Complete Implementation

The complete, working Petfinder MCP server we'll be building is available at: https://github.com/mattlgroff/petfinder-mcp-server

This repository includes the full source code, setup instructions, and deployment examples you can reference as we build each component together.

12 Petfinder's Toolbox: Different Tools, Same Patterns

Now that you understand how MCP tools work from our weather example, let's see how the same patterns apply to a different domain: adoptable pets. The structure remains identical, but the tools serve a completely different purpose.

12.1 The Petfinder Toolbox

When an AI agent asks our Petfinder server "what tools do you have?" via `tools/list`, we respond with our pet-focused toolkit:


```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "tools": [
      {
        "name": "searchAnimals",
        "title": "Search for adoptable pets",
        "description": "Find adoptable dogs, cats, and other pets by location, breed, age, and more",
        "inputSchema": {
          "type": "object",
          "properties": {
            "type": {
              "type": "string",
              "description": "Animal type (dog, cat, rabbit, bird, etc.)",
              "default": "dog"
            },
            "location": {
              "type": "string",
              "description": "ZIP code, city, or state to search in"
            },
            "breed": {
              "type": "string",
              "description": "Specific breed to search for"
            },
            "age": {
              "type": "string",
              "enum": ["baby", "young", "adult", "senior"],
              "description": "Age category of the pet"
            },
            "size": {
              "type": "string",
              "enum": ["small", "medium", "large", "xlarge"],
              "description": "Size of the pet"
            },
            "limit": {
              "type": "integer",
              "minimum": 1,
              "maximum": 100,
              "default": 20,
              "description": "Maximum number of results to return"
            }
          },
          "required": ["location"]
        }
      },
      {
        "name": "getAnimalDetails",
        "title": "Get detailed pet information",
        "description": "Get complete details about a specific adoptable pet including photos, contact info, an
        "inputSchema": {
          "type": "object",
          "properties": {
            "animalId": {
              "type": "string",
              "description": "The unique ID of the pet from search results"
            }
          },
          "required": ["animalId"]
        }
      },
      {
        "name": "searchOrganizations",
        "title": "Find animal shelters and rescues",
        "description": "Locate animal shelters, rescue organizations, and adoption centers in your area",
        "inputSchema": {
```

```
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "ZIP code, city, or state to search in"
            },
            "name": {
              "type": "string",
              "description": "Name or partial name of the organization"
            },
            "limit": {
              "type": "integer",
              "minimum": 1,
              "maximum": 100,
              "default": 20,
              "description": "Maximum number of results to return"
            }
          },
          "required": ["location"]
        }
      }
    ]
  }
}
```

12.2 Comparing Domains: Weather vs Pets

Notice how the structure is identical to our weather tools, but the content is completely different:

12.2.1 Weather Tools

- `getCurrentWeather` - Real-time conditions
- `getWeatherForecast` - Future predictions
- `searchLocations` - Find places

12.2.2 Pet Tools

- `searchAnimals` - Find adoptable pets
- `getAnimalDetails` - Pet information
- `searchOrganizations` - Find shelters

The pattern is the same: search, get details, find related entities.

12.3 The Zod Implementation

Since we already understand Zod from the weather example, here's our Petfinder schemas:

```
import { z } from 'zod';

const searchAnimalsSchema = z.object({
  type: z.string().default('dog').describe('Animal type (dog, cat, rabbit, bird, etc.)'),
  location: z.string().describe('ZIP code, city, or state to search in'),
  breed: z.string().optional().describe('Specific breed to search for'),
  age: z.enum(['baby', 'young', 'adult', 'senior']).optional().describe('Age category of the pet'),
  size: z.enum(['small', 'medium', 'large', 'xlarge']).optional().describe('Size of the pet'),
  limit: z.number().int().min(1).max(100).default(20).describe('Maximum number of results to return'),
});

const getAnimalDetailsSchema = z.object({
  animalId: z.string().describe('The unique ID of the pet from search results'),
});

const searchOrganizationsSchema = z.object({
  location: z.string().describe('ZIP code, city, or state to search in'),
  name: z.string().optional().describe('Name or partial name of the organization'),
  limit: z.number().int().min(1).max(100).default(20).describe('Maximum number of results to return'),
});

const tools = [
  {
    name: 'searchAnimals',
    title: 'Search for adoptable pets',
    description: 'Find adoptable dogs, cats, and other pets by location, breed, age, and more',
    inputSchema: zodToMCPSchema(searchAnimalsSchema),
  },
```

```
  {
    name: 'getAnimalDetails',
    title: 'Get detailed pet information',
    description: 'Get complete details about a specific adoptable pet including photos, contact info, and adop
    inputSchema: zodToMCPSchema(getAnimalDetailsSchema),
  },
  {
    name: 'searchOrganizations',
    title: 'Find animal shelters and rescues',
    description: 'Locate animal shelters, rescue organizations, and adoption centers in your area',
    inputSchema: zodToMCPSchema(searchOrganizationsSchema),
  }
];
```

12.4 What Makes These Tools Different

12.4.1 Complex Search Parameters

Unlike weather (just location), pet searches have many optional filters: - Animal type (dog, cat, rabbit, etc.) - Breed (Golden Retriever, Persian, etc.) - Age category (baby, young, adult, senior) - Size (small, medium, large, xlarge)

12.4.2 Hierarchical Data

Pets belong to organizations, creating natural relationships: 1. Search animals → Get list of pets 2. Get animal details → Full information about one pet 3. Search organizations → Find shelters and rescues

12.4.3 Rich Media Content

Pet data includes photos, descriptions, and contact information - much richer than weather data.

12.5 The Handler Implementation

```
async function handleMCPRequest(request: MCPRequest): Promise<MCPResponse | null> {
  switch (request.method) {
    case 'initialize':
      return {
        jsonrpc: '2.0',
        id: request.id!,
        result: {
          protocolVersion: '2024-11-05',
          capabilities: {
            tools: { listChanged: true },
          },
          serverInfo: {
            name: 'petfinder-mcp-server',
            version: '1.0.0',
          },
        },
      };

    case 'tools/list':
      return {
        jsonrpc: '2.0',
        id: request.id!,
        result: { tools },
      };

    case 'tools/call':
      // OAuth authentication complexity comes in the next chapters
      return await handleToolCall(request);

    default:
      return {
        jsonrpc: '2.0',
        id: request.id || 'unknown',
        error: {
          code: -32601,
          message: `Method ${request.method} not found`,
        },
      };
  }
}
```

12.6 What's Different About Petfinder

The biggest difference isn't in the tools themselves - it's in the authentication that happens behind the scenes. When an AI agent calls `searchAnimals`, our server needs to:

1. **Check our token** - Do we have a valid access token?
2. **Refresh if needed** - Is it expired? Get a new one
3. **Make the API call** - Use Bearer token authentication
4. **Handle errors** - OAuth errors are different from API key errors

The tools look the same to the AI agent, but the authentication complexity is hidden in our implementation.

12.7 Real-World Usage Scenarios

These tools enable AI agents to help users with pet adoption:

User: "I'm looking for a medium-sized dog in San Francisco" **AI Agent**: Uses `searchAnimals` with type=dog, size=medium, location="San Francisco"

User: "Tell me more about this golden retriever" **AI Agent**: Uses `getAnimalDetails` with the specific pet ID

User: "What shelters are near me?" **AI Agent**: Uses `searchOrganizations` with the user's location

12.8 The Authentication Preview

In our weather server, every API call looked like:

```
const response = await fetch(`http://api.weatherapi.com/v1/current.json?key=${API_KEY}&q=${location}`);
```

In our Petfinder server, it will look like:

```
const token = await getValidAccessToken(); // Complex OAuth logic here
const response = await fetch('https://api.petfinder.com/v2/animals', {
  headers: {
    'Authorization': `Bearer ${token}`,
  }
});
```

That `getValidAccessToken()` function contains all the OAuth complexity we'll implement in the next chapters.

12.9 Same Patterns, Different Domains

The beauty of MCP is that once you understand the pattern with weather data, you can apply it to any domain: - **E-commerce**: search products, get details, find stores - **News**: search articles, get content, find sources
- **Travel**: search flights, get itineraries, find hotels - **Finance**: get stock prices, search companies, analyze portfolios

The tools change, but the MCP protocol remains the same.

13 Petfinder API Wrappers: Same Structure, Different Authentication

The API wrapper pattern you learned with WeatherAPI applies directly to Petfinder, but with one crucial difference: OAuth token management. Let's build the same three-layer architecture with token-based authentication.

13.1 The Three Layers (Redux)

Just like our weather server:

1. **MCP Layer**: Handles JSON-RPC communication with AI agents
2. **Validation Layer**: Uses Zod schemas to ensure inputs are correct
3. **API Layer**: Makes HTTP requests to Petfinder (but with Bearer tokens)

13.2 The Petfinder API Wrapper Function

Here's our core function - notice how similar it is to the weather version:

```
const PETFINDER_API_BASE = 'https://api.petfinder.com/v2';

async function callPetfinderAPI(endpoint: string, accessToken: string, params: Record<string, any> = {}) {
  // Build the URL
  const url = new URL(`${PETFINDER_API_BASE}${endpoint}`);

  // Add query parameters
  Object.entries(params).forEach(([key, value]) => {
    if (value !== undefined && value !== null) {
      url.searchParams.set(key, value.toString());
    }
  });

  // Make the HTTP request with Bearer token
  const response = await fetch(url.toString(), {
    method: 'GET',
    headers: {
      'Authorization': `Bearer ${accessToken}`,
      'Content-Type': 'application/json',
    },
  });

  if (!response.ok) {
    const errorText = await response.text();

    let errorBody;
    try {
      errorBody = JSON.parse(errorText);
    } catch {
      errorBody = { message: errorText };
    }

    throw new PetfinderAPIError(
      response.status,
      response.statusText,
      errorBody,
      `Petfinder API request failed: ${errorText}`
    );
  }

  return response.json();
}

class PetfinderAPIError extends Error {
  constructor(
    public status: number,
    public statusText: string,
    public body: any,
    message?: string
  ) {
    super(message || `Petfinder API error (${status}): ${statusText}`);
    this.name = 'PetfinderAPIError';
  }
}
```

13.3 Key Differences from WeatherAPI

13.3.1 1. Authorization Header Instead of Query Parameter

WeatherAPI:

```
url.searchParams.set('key', apiKey);  // API key in URL
```

Petfinder:

```
headers: {
  'Authorization': `Bearer ${accessToken}`,  // Token in header
}
```

13.3.2 2. Token Management

The weather server used a static API key. The Petfinder server needs dynamic token management:

```
async function callPetfinderAPI(endpoint: string, accessToken: string, params: Record<string, any> = {}) {
  // The accessToken parameter comes from our OAuth token manager
  // (We'll implement this in the next chapter)
}
```

13.4 Individual API Wrapper Functions

```
// Search for adoptable animals
async function searchAnimalsData(accessToken: string, params: z.infer<typeof searchAnimalsSchema>) {
  return await callPetfinderAPI('/animals', accessToken, {
    type: params.type,
    location: params.location,
    breed: params.breed,
    age: params.age,
    size: params.size,
    limit: params.limit,
  });
}

// Get detailed information about a specific animal
async function getAnimalDetailsData(accessToken: string, params: z.infer<typeof getAnimalDetailsSchema>) {
  return await callPetfinderAPI(`/animals/${params.animalId}`, accessToken);
}

// Search for animal organizations (shelters, rescues)
async function searchOrganizationsData(accessToken: string, params: z.infer<typeof searchOrganizationsSchema>)
  return await callPetfinderAPI('/organizations', accessToken, {
    location: params.location,
    name: params.name,
    limit: params.limit,
  });
}
```

These functions look almost identical to the weather ones - just different endpoint paths and parameter names.

13.5 MCP Tool Implementations

```
async function searchAnimals(accessToken: string, input: any) {
  // Apply defaults and validate
  const inputWithDefaults = {
    type: 'dog',
    limit: 20,
    ...input
  };
  const validatedInput = searchAnimalsSchema.parse(inputWithDefaults);

  // Make the API call
  const result = await searchAnimalsData(accessToken, validatedInput);

  // Format the response for the AI agent
  const animals = result.animals || [];
  let summary = `Found ${animals.length} adoptable ${validatedInput.type}s in ${validatedInput.location}:\n\n`

  animals.forEach((animal: any) => {
```

```
    summary += `🐕 ${animal.name} - ${animal.breeds.primary}\n`;
    summary += `   Age: ${animal.age} | Size: ${animal.size} | Gender: ${animal.gender}\n`;
    summary += `   Status: ${animal.status} | ID: ${animal.id}\n`;
    if (animal.description) {
      summary += `   ${animal.description.substring(0, 100)}...\n`;
    }
    summary += `\n`;
  });

  return {
    content: [
      {
        type: 'text',
        text: summary,
      },
      {
        type: 'text',
        text: JSON.stringify(result, null, 2),
      },
    ],
  };
}

async function getAnimalDetails(accessToken: string, input: any) {
  // Validate the input
  const validatedInput = getAnimalDetailsSchema.parse(input);

  // Make the API call
  const result = await getAnimalDetailsData(accessToken, validatedInput);

  // Format the response
  const animal = result.animal;
  let summary = `🐾 ${animal.name} - Detailed Information\n\n`;
  summary += `Breed: ${animal.breeds.primary}${animal.breeds.secondary ? ` / ${animal.breeds.secondary}` : ''}
  summary += `Age: ${animal.age} | Size: ${animal.size} | Gender: ${animal.gender}\n`;
  summary += `Colors: ${animal.colors.primary}${animal.colors.secondary ? ` / ${animal.colors.secondary}` : ''
  summary += `Spayed/Neutered: ${animal.attributes.spayed_neutered ? 'Yes' : 'No'}\n`;
  summary += `House Trained: ${animal.attributes.house_trained ? 'Yes' : 'Unknown'}\n`;
  summary += `Good with: `;
  if (animal.attributes.good_with_children) summary += 'Children ';
  if (animal.attributes.good_with_dogs) summary += 'Dogs ';
  if (animal.attributes.good_with_cats) summary += 'Cats ';
  summary += `\n\n`;

  if (animal.description) {
    summary += `Description: ${animal.description}\n\n`;
  }

  if (animal.contact?.email || animal.contact?.phone) {
    summary += `Contact Information:\n`;
    if (animal.contact.email) summary += `Email: ${animal.contact.email}\n`;
    if (animal.contact.phone) summary += `Phone: ${animal.contact.phone}\n`;
    if (animal.contact.address) {
      summary += `Address: ${animal.contact.address.address1}, ${animal.contact.address.city}, ${animal.contac
    }
  }

  return {
    content: [
      {
        type: 'text',
        text: summary,
      },
      {
        type: 'text',
        text: JSON.stringify(result, null, 2),
      },
    ],
  };
}

async function searchOrganizations(accessToken: string, input: any) {
  // Apply defaults and validate
  const inputWithDefaults = {
    limit: 20,
    ...input
  };
  const validatedInput = searchOrganizationsSchema.parse(inputWithDefaults);

  // Make the API call
  const result = await searchOrganizationsData(accessToken, validatedInput);

  // Format the response
  const organizations = result.organizations || [];
```

```
  let summary = `Found ${organizations.length} animal organizations in ${validatedInput.location}:\n\n`;

  organizations.forEach((org: any) => {
    summary += `🏠 ${org.name}\n`;
    summary += `   Email: ${org.email || 'Not provided'}\n`;
    summary += `   Phone: ${org.phone || 'Not provided'}\n`;
    if (org.address) {
      summary += `   Address: ${org.address.address1}, ${org.address.city}, ${org.address.state}\n`;
    }
    if (org.website) {
      summary += `   Website: ${org.website}\n`;
    }
    summary += `\n`;
  });

  return {
    content: [
      {
        type: 'text',
        text: summary,
      },
      {
        type: 'text',
        text: JSON.stringify(result, null, 2),
      },
    ],
  };
}
```

13.6 The Tool Registry

```
const allTools = {
  'searchAnimals': searchAnimals,
  'getAnimalDetails': getAnimalDetails,
  'searchOrganizations': searchOrganizations,
};
```

13.7 What's Different About the Data

13.7.1 Rich Pet Information

Unlike weather data, pet records contain: - Multiple photos - Detailed behavioral attributes - Contact information for adoption - Shelter/rescue organization details

13.7.2 Hierarchical Relationships

- Animals belong to organizations
- Breeds have primary/secondary classifications
- Attributes are boolean flags (good_with_children, house_trained, etc.)

13.7.3 Real-World Impact

This isn't just data - it's about connecting pets with families. The formatting emphasizes actionable information like contact details and adoption status.

13.8 The Missing Piece: Token Management

Notice that all our functions require an `accessToken` parameter. In the weather server, we just passed the API key directly. But where does the access token come from?

That's the OAuth complexity we'll solve in the next chapter. We need a token manager that:

1. **Exchanges** client credentials for access tokens
2. **Caches** tokens in memory to avoid repeated requests
3. **Refreshes** expired tokens automatically
4. **Handles** OAuth errors gracefully

13.9 Same Patterns, OAuth Complexity

The beauty of this approach is that the API wrapper functions don’t care about OAuth complexity. They just expect a valid access token. All the OAuth logic is hidden in a separate token management layer.

This separation of concerns makes the code easier to understand, test, and maintain - the same architectural principles that make MCP servers so powerful.

14 OAuth Authentication Strategy: The Real Difference

Now we arrive at the heart of what makes Petfinder different from WeatherAPI: OAuth 2.0 token management. This chapter focuses on the authentication complexity that transforms our simple API key pattern into a sophisticated token management system.

14.1 The Authentication Gap

Let's compare what we've built so far:

14.1.1 WeatherAPI (Simple)

```
// One line of authentication
url.searchParams.set('key', apiKey);
```

14.1.2 Petfinder (Complex)

```
// Multi-step OAuth flow
const token = await getAccessToken(clientId, clientSecret);
headers: { 'Authorization': `Bearer ${token}` }
```

That `getAccessToken()` function contains all the OAuth complexity. Let's build it step by step.

14.2 OAuth 2.0 Client Credentials Flow

Petfinder uses the "client credentials" OAuth flow, which works like this:

1. **Exchange Credentials** → You send `client_id` + `client_secret` to get an `access_token`
2. **Use Token** → You use the `access_token` in API calls until it expires
3. **Refresh Token** → When expired, repeat step 1 to get a new token

```
// Step 1: Exchange credentials for token
POST https://api.petfinder.com/v2/oauth2/token
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&client_id=YOUR_ID&client_secret=YOUR_SECRET

// Response:
{
  "token type": "Bearer",
  "expires in": 3600,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOi..."
}

// Step 2: Use token in API calls
GET https://api.petfinder.com/v2/animals
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOi...
```

14.3 Token Cache Implementation

Since tokens last 3600 seconds (1 hour), we need to cache them in memory:

```
interface TokenCache {
  access token: string;
  expires_at: number; // Unix timestamp
}

// Per-client token cache - supports multiple clients
const tokenCache = new Map<string, TokenCache>();
```

The key insight: **each client ID gets its own cached token**. This allows one MCP server to handle multiple Petfinder developer accounts simultaneously.

Note: Alternatively you could make in the CLIENT_ID and CLIENT_SECRET as environmental variables and secure your MCP Server via private vpc or publically via an additional secret passed in via headers. Whatever works best for your situation.

14.4 The Token Manager

Here's the core token management logic:

```
async function getAccessToken(clientId: string, clientSecret: string): Promise<string> {
  const now = Math.floor(Date.now() / 1000);

  // Validate credentials
  if (!clientId || !clientSecret) {
    throw new Error('Missing Petfinder credentials');
  }

  // Check cache first - with 60-second buffer
  const cachedToken = tokenCache.get(clientId);
  if (cachedToken && cachedToken.expires_at > now + 60) {
    console.log(`🔄 Using cached token for client: ${clientId}`);
    return cachedToken.access_token;
  }

  // Request new token
  console.log('🔄 Requesting new Petfinder access token...');
  const response = await fetch(`${PETFINDER_BASE}/oauth2/token`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
    },
    body: new URLSearchParams({
      grant type: 'client credentials',
      client id: clientId,
      client_secret: clientSecret,
    }),
  });

  if (!response.ok) {
    const errorText = await response.text();
    throw new PetfinderAPIError(
      response.status,
      response.statusText,
      errorText,
      `OAuth token request failed: ${errorText}`
    );
  }

  // Parse and cache the new token
  const tokenData = await response.json();
  const newToken: TokenCache = {
    access_token: tokenData.access_token,
    expires_at: now + tokenData.expires_in,
  };

  tokenCache.set(clientId, newToken);
  console.log(`✅ New token obtained, expires in ${tokenData.expires_in} seconds`);

  return newToken.access_token;
}
```

14.5 Key Design Decisions

14.5.1 1. 60-Second Buffer

```
if (cachedToken && cachedToken.expires_at > now + 60)
```

We refresh tokens 60 seconds before they expire to avoid race conditions where a token expires mid-request.

14.5.2 2. Per-Client Caching

```
const tokenCache = new Map<string, TokenCache>();
tokenCache.set(clientId, newToken);
```

Each `client_id` gets its own cache entry, allowing multiple Petfinder accounts to use the same MCP server.

14.5.3 3. Form-Encoded OAuth Request

```
 headers: {
   'Content-Type': 'application/x-www-form-urlencoded',
 },
 body: new URLSearchParams({
   grant_type: 'client_credentials',
   client_id: clientId,
   client_secret: clientSecret,
 })
```

OAuth token requests use form encoding, not JSON. This is part of the OAuth 2.0 specification.

14.6 Request Context Pattern

Our API wrapper functions need access to credentials, but we don't want to pass them through every function call. The solution: request context.

```
 // Global request context
 let requestContext: { clientId?: string; clientSecret?: string } = {};

 async function petfinderRequest(endpoint: string, params?: Record<string, any>) {
   const { clientId, clientSecret } = requestContext;

   if (!clientId || !clientSecret) {
     throw new Error('Request context missing credentials');
   }

   // Get valid token using context credentials
   const token = await getAccessToken(clientId, clientSecret);

   // Make API call with Bearer token
   const response = await fetch(url, {
     headers: {
       'Authorization': `Bearer ${token}`,
       'Content-Type': 'application/json',
     },
   });

   // Handle response...
 }
```

14.7 Credential Extraction

Unlike WeatherAPI's `Authorization: Bearer api-key` header, Petfinder credentials come via query parameters:

```
 function extractCredentialsFromQuery(url: URL): { clientId?: string; clientSecret?: string } {
   const clientId = url.searchParams.get('client-id') || undefined;
   const clientSecret = url.searchParams.get('client-secret') || undefined;

   return { clientId, clientSecret };
 }
```

This allows MCP clients to pass credentials in the server URL:

```
http://localhost:3000/mcp?client-id=your-id&client-secret=your-secret
```

Note: The reason I set this up this way is for MCP Inspector and more MCP Client support. Claude.ai and MCP Inspector didn't allow sending custom headers that were not Authorization. So it was simpler for me to pass them as query parameters instead.

14.8 The Complete Flow

Here's how authentication works in a complete `tools/call` request:

```
case 'tools/call':
  // 1. Extract credentials from URL
  const { clientId, clientSecret } = extractCredentialsFromQuery(url);

  // 2. Validate credentials are present
  if (!clientId || !clientSecret) {
    return createErrorResponse(id, -32001, 'Authentication required');
  }

  try {
    // 3. Set credentials in request context
    requestContext = { clientId, clientSecret };

    // 4. Call tool (which will use getAccessToken internally)
    const result = await handler(args);

    return { jsonrpc: '2.0', id, result };
  } finally {
    // 5. Clear credentials after request
    requestContext = {};
  }
```

14.9 Cache Cleanup

To prevent memory leaks, we periodically clean up expired tokens:

```
function cleanupExpiredTokens() {
  const now = Math.floor(Date.now() / 1000);

  for (const [clientId, token] of tokenCache.entries()) {
    if (token.expires at <= now) {
      tokenCache.delete(clientId);
    }
  }
}
```

This runs every time we request a new token, keeping memory usage bounded.

14.10 Error Handling

OAuth introduces new error types:

```
 case 400: return -32602; // Invalid credentials format
case 401: return -32001; // Invalid client_id/client_secret
case 403: return -32002; // Client not authorized for grants
```

These map to JSON-RPC error codes that help AI agents understand what went wrong.

14.11 Comparing Authentication Strategies

14.11.1 WeatherAPI Pattern (Static API Key)

- One API key for everything
- No expiration to manage
- Simple header: `Authorization: Bearer api-key`

14.11.2 Petfinder Pattern (OAuth Client ID and Client Secret)

- Complex token management
- Network requests for token refresh
- In-memory state to manage

14.12 The Hidden Complexity

From the AI agent's perspective, both servers look identical:

```
{
  "jsonrpc": "2.0",
  "method": "tools/call",
  "params": {
    "name": "searchAnimals",
    "arguments": { "type": "dog", "location": "90210" }
  }
}
```

But behind the scenes: - **WeatherAPI**: Direct API call with static key - **Petfinder**: Token cache check → possible OAuth exchange → API call with Bearer token

This demonstrates the power of good abstractions: complex authentication logic hidden behind a simple interface.

14.13 Why This Matters

Many modern APIs use OAuth 2.0 instead of simple API keys: - **Google APIs** (Gmail, Calendar, Drive) - **Microsoft Graph** (Office 365, Teams) - **Slack API** - **Twitter API v2** - **GitHub API** (for user data)

Understanding this pattern prepares you to wrap any OAuth-protected API with MCP servers, even if they approach things a bit different the concept will apply.

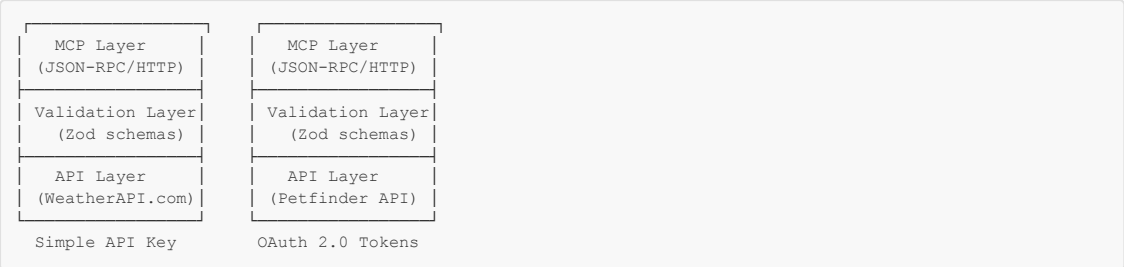
The authentication strategy is what makes the difference between a simple proof-of-concept and a production-ready MCP server that can handle real-world API complexity.

15 Comparing the Two Approaches: Key Takeaways

Now that we've built two complete MCP servers, let's step back and examine what we've learned. The WeatherAPI and Petfinder examples demonstrate two different authentication patterns that cover the majority of APIs you'll encounter.

15.1 Architecture Comparison

Both servers follow the same three-layer architecture:



The top two layers are nearly identical. The difference lies in the API layer's authentication strategy.

15.2 Authentication Patterns

15.2.1 Pattern 1: API Key Authentication (WeatherAPI)

Implementation:

```
// Simple and direct
const response = await fetch(`${API_BASE}/endpoint?key=${apiKey}&param=${value}`);
```

15.2.2 Pattern 2: OAuth 2.0 Client Credentials (Petfinder)

Implementation:

```
// Multi-step process
const token = await getAccessToken(clientId, clientSecret);
const response = await fetch(`${API_BASE}/endpoint`, {
  headers: { 'Authorization': `Bearer ${token}` },
  body: JSON.stringify(params)
});
```

15.3 The Universal Pattern

Despite their authentication differences, both servers demonstrate the universal MCP pattern:

1. **Define your tools** with clear schemas
2. **Validate inputs** using Zod or similar
3. **Call external APIs** with proper authentication
4. **Format responses** for AI agent consumption
5. **Handle errors** gracefully with descriptive messages

The authentication layer is simply determined by the API you're wrapping - you implement whatever authentication method the target API requires. The core MCP concepts remain the same regardless of the authentication complexity.

15.4 What We've Accomplished

Through these two examples, we've covered the fundamental patterns you'll encounter when building MCP servers. The authentication layer is simply determined by the API you're wrapping - you implement whatever authentication method the target API requires, while the core MCP concepts remain consistent across all implementations.

Whether you're integrating with simple data APIs or complex enterprise systems, the three-layer architecture and JSON-RPC patterns we've explored will serve as your foundation for any MCP server you build.

16 Conclusion: You Did It!

Congratulations! You've just built two complete MCP servers using fundamentally different authentication strategies. That's no small accomplishment - you've mastered the concepts that will let you integrate any REST API with AI agents.

16.1 What You've Mastered

Through our journey together, you've learned:

- **The MCP Protocol** - JSON-RPC communication between AI agents and your servers
- **Tool Definition** - How to expose API functions as discoverable tools
- **Input Validation** - Using Zod schemas to ensure data quality and provide helpful errors
- **API Key Authentication** - The simple, direct approach used by many services
- **OAuth 2.0 Flow** - Complex token management with caching and refresh logic
- **Error Handling** - Graceful degradation and informative error messages
- **Response Formatting** - Structuring data for AI agent consumption

Most importantly, you understand the **universal MCP pattern** that applies regardless of which API you're wrapping.

16.2 You're Ready to Build MCP Servers

Armed with these two patterns, you can now wrap virtually any REST API:

API Key APIs: News APIs, currency rates, basic web services, public data sources **OAuth APIs**: Google services, Microsoft Graph, Slack, Twitter, enterprise systems

The patterns scale from simple data APIs to complex enterprise integrations, giving you the foundation to build MCP servers for any domain you need to integrate with AI agents.

16.3 Complete Source Code

Both example servers we built in this book are available as complete, working implementations:

WeatherAPI MCP Server: https://github.com/mattlgroff/weather-api-mcp-server **Petfinder MCP Server**: https://github.com/mattlgroff/petfinder-mcp-server

These repositories include the complete source code, setup instructions, and deployment examples you can use as starting points for your own MCP servers. They're both single-file implementations that are perfect for passing into LLM chatbots or agentic coding assistants like Cursor or OpenCode.

16.4 Let's Stay Connected

Building MCP servers is just the beginning. The AI agent ecosystem is evolving rapidly, and I'd love to help you on your journey.

Have questions? Email me directly at **matt@groff.dev** - I read every message and love helping developers build with AI.

Want to connect? Find me on LinkedIn at https://www.linkedin.com/in/mattgroff/ where I share updates about AI development and new projects.

Looking for more content? Check out my blog at **groff.dev** where I regularly write about web development, AI integration, and emerging technologies.

16.5 Thank You

Thank you for supporting me by reading this book. Your time is valuable, and I'm honored you chose to spend it learning about MCP servers with me.

I hope this book has equipped you with the knowledge and confidence to build your own MCP servers. The world of AI agents is expanding rapidly, and the tools you create will empower them to do incredible things.

Now go build something amazing. The AI agents are waiting for the tools only you can provide.