Complete Guide to Notion MCP Server System
From the July AI Team
Teaching Guide
July 17, 2025

Contents

1 Introduction: What is the Notion MCP Server?

This document will teach you how to understand and use a system that lets you chat with an AI (Claude) to manage your Notion workspace through natural conversation. Think of it as having a personal assistant that can create pages, search your notes, organize databases, and help you manage your Notion content without ever opening the Notion app.

1.1 What Makes This Special?

Traditional approach: You open Notion, navigate through pages, click buttons, and manually create content.

Our approach: You simply tell Claude "Create a meeting notes page in my project workspace" or "Show me all pages tagged with 'urgent'" and it happens automatically.

1.2 How the System Works (Overview)

You Chat with Claude → Claude Uses MCP Tools → Notion Updates → You Get Results

The magic happens through something called MCP (Model Context Protocol) - a way for AI to safely interact with Notion and other external systems.

2 Understanding the Technology Stack

Before diving into the code, let's understand what technologies we're using and why:

2.1 The Components

- **Notion API:** The official way to programmatically interact with Notion workspaces
- **Node.js:** The programming language/environment our server runs in
- **MCP Server:** The "translator" that converts Claude's requests into Notion operations
- **Claude Desktop:** The AI interface you chat with

2.2 Why These Choices?

Notion API over screen automation:

- Direct, reliable access to your data
- Faster than clicking through interfaces
- Access to all Notion features programmatically
- Official support and documentation

Node.js over other languages:

- Excellent for handling real-time communication
- Huge ecosystem of packages

- Same language (JavaScript) used in web browsers
- Great JSON handling (Notion API uses JSON)

3 Understanding Notion's Data Structure

3.1 How Notion Organizes Information

Notion has a hierarchical structure:

- **Workspaces:** Top-level containers (like your entire filing cabinet)
- **Pages:** Individual documents that can contain text, databases, or other content
- **Databases:** Structured collections of information (like spreadsheets with superpowers)
- **Blocks:** Individual pieces of content within pages (paragraphs, headings, lists, etc.)

3.2 Why This Design Matters

Understanding this hierarchy helps you use our MCP server effectively:

- Creating a page requires knowing which parent page or database it belongs to
- Reading content means fetching both page properties and the blocks within it
- Databases have schemas that define what types of information they can store

Real-world analogy: Think of it like organizing a physical office:

- Workspace = Your entire office building
- Pages = Individual file folders
- Databases = Filing cabinets with labeled drawers
- Blocks = Individual documents within folders

4 Understanding JavaScript Basics in This Code

4.1 Imports and Dependencies

At the beginning of our server code, you'll see:

```
import { Server } from "@modelcontextprotocol/sdk/server/index.js";
import { Client } from "@notionhq/client";

const notion = new Client({
  auth: process.env.NOTION_API_KEY,
});
```

Understanding Imports: Think of imports like borrowing tools from a toolbox. We're saying "I want to use the Notion tools from this specific package."

Environment Variables: `process.env.NOTION_API_KEY` reads a secret value from the system's environment. It's like having a locked drawer where you keep passwords - the code can access them, but they're not visible in the code itself.

4.2 Async/Await Pattern

Throughout the code, you'll see:

```
async function createPage(args) {
  const response = await notion.pages.create({
    parent: { page_id: args.parent_id },
    properties: {
      title: {
        title: [{ text: { content: args.title } }]
      }
    }
  });

  return response;
}
```

Why async/await? Notion API operations take time (like mailing a letter and waiting for a reply). async/await lets our program do other things while waiting, instead of freezing up.

Real-world analogy: It's like ordering food at a restaurant:

- `async` = "This might take a while"
- `await` = "Wait for the food before continuing to eat"
- Without await = "Start eating before the food arrives" (doesn't work!)

5 Breaking Down the MCP Server Code

5.1 Server Setup and Communication

```
const server = new Server(
  {
    name: "notion-mcp-server",
    version: "1.0.0",
  },
  {
    capabilities: {
      tools: {},
      resources: {},
    },
  }
);
```

What's happening here?

Server initialization: This creates our MCP server with a name and capabilities. Think of it like setting up a phone system - we're telling Claude what our server can do.

Capabilities: We declare that we support "tools" (actions Claude can ask us to perform) and "resources" (information Claude can read from us).

5.2 Request Handling Logic

```
server.setRequestHandler(CallToolRequestSchema, async (request) => {
  const { name, arguments: args } = request.params;

  try {
    switch (name) {
      case "search_pages":
        const result = await searchPages(args);
        return { content: [{ type: "text", text: result }] };

      case "create_page":
        const response = await createPage(args);
        return { content: [{ type: "text", text: response }] };

      default:
        throw new Error(`Unknown tool: ${name}`);
    }
  } catch (error) {
    return {
      content: [{ type: "text", text: `Error: ${error.message}` }],
      isError: true,
    };
  }
});
```

Understanding the Flow:

1. **Receive request:** Claude sends us a tool call
2. **Extract parameters:** We get the tool name and arguments
3. **Route to handler:** We call the appropriate function
4. **Return response:** We send results back to Claude
5. **Error handling:** If anything goes wrong, we send an error message

5.3 Tool Definition and Schema

```
{
  name: "create_page",
  description: "Create a new page in Notion",
  inputSchema: {
    type: "object",
    properties: {
      parent_id: {
        type: "string",
        description: "Parent page or database ID"
      },
      title: {
        type: "string",
        description: "Page title"
      },
      content: {
        type: "string",
```

```
        description: "Page content (optional)"
      }
    },
    required: ["parent_id", "title"]
  }
}
```

Why define schemas?
Type safety: Claude knows exactly what parameters each tool expects.
Validation: Invalid requests are caught before reaching Notion.
Documentation: The descriptions help Claude understand how to use each tool.
Real-world analogy: It's like a restaurant menu:

- `name` = Dish name ("Caesar Salad")
- `description` = What it is ("Fresh romaine with parmesan")
- `inputSchema` = Options ("Dressing on side? Add chicken?")
- `required` = Mandatory choices ("Choose your protein")

6 Notion Operations: CRUD in Action

6.1 Create Operations

```
async function createPage(args) {
  const { parent_id, title, content } = args;

  if (!title) {
    throw new Error('Title is required');
  }

  const properties = {
    title: {
      title: [{ text: { content: title } }]
    }
  };

  const children = [];
  if (content) {
    children.push({
      object: "block",
      type: "paragraph",
      paragraph: {
        rich_text: [{ type: "text", text: { content: content } }]
      }
    });
  }

  const response = await notion.pages.create({
    parent: { page_id: parent_id },
    properties,
    children
```

```
  });

  return `Page created successfully with ID: ${response.id}`;
}
```

Breaking down this function:

Destructuring: `const {parent_id, title, content} = args` extracts properties from the arguments object.

Validation: We check required fields before attempting Notion operations.

Notion structure: Pages in Notion have properties (metadata) and children (content blocks).

Error handling: If the Notion operation fails, we throw an error that gets caught by our request handler.

6.2 Read Operations

```
async function searchPages(args) {
  const { query, page_size = 10 } = args;

  const searchParams = { page_size };

  if (query) {
    searchParams.query = query;
  }

  const response = await notion.search(searchParams);

  return JSON.stringify(response, null, 2);
}
```

Query building pattern:

1. **Start with base parameters:** Set default page size
2. **Add filters conditionally:** Only add query if provided
3. **Execute search:** Call Notion's search API
4. **Format response:** Convert to readable JSON

Why build queries this way? This pattern lets us handle optional parameters elegantly. Whether someone asks for "all pages" vs "pages about project planning", the same function handles both cases.

6.3 Update Operations

```
async function updatePage(args) {
  const { page_id, title, content } = args;

  if (!page_id) {
    throw new Error('Page ID is required');
  }

  const properties = {};
```

```
  if (title) {
    properties.title = {
      title: [{ text: { content: title } }]
    };
  }

  const response = await notion.pages.update({
    page_id,
    properties
  });

  // Update content blocks if provided
  if (content) {
    // First, get existing blocks
    const existingBlocks = await notion.blocks.children.list({
      block_id: page_id
    });

    // Delete existing blocks
    for (const block of existingBlocks.results) {
      await notion.blocks.delete({ block_id: block.id });
    }

    // Add new content
    await notion.blocks.children.append({
      block_id: page_id,
      children: [{
        object: "block",
        type: "paragraph",
        paragraph: {
          rich_text: [{ type: "text", text: { content: content } }]
        }
      }]
    });
  }

  return `Updated page: "${response.properties.title.title[0].text.content
     }" (ID: ${response.id})`;
}
```

The two-step update process:

Step 1: Update page properties (like title) using the pages API.

Step 2: Update page content by replacing blocks using the blocks API.

Why separate steps? Notion treats page metadata and content as different things, requiring different API calls.

7 Error Handling and Robustness

7.1 Defensive Programming Patterns

```
// 1. Input validation
if (!title) {
```

```
  throw new Error('Title is required');
}

// 2. Notion API error handling
const { data, error } = await notion.pages.create(pageData);
if (error) throw error;

// 3. Try-catch at the request level
try {
  let result = await createPage(args);
  sendResponse(result);
} catch (error) {
  sendError(request.id, -32603, `Tool execution failed: ${error.message}`)
      ;
}
```

Three levels of error handling:

1. **Input validation:** Check requirements before doing work
2. **API errors:** Handle Notion-specific problems
3. **Request-level catches:** Ensure we always send a response to Claude

Why this matters: Without proper error handling, a single typo or network hiccup could crash the entire system. This approach ensures Claude always gets a response, even if it's "something went wrong."

7.2 Response Format Consistency

```
// Success response
return {
  content: [{ type: "text", text: result }]
};

// Error response
return {
  content: [{ type: "text", text: `Error: ${error.message}` }],
  isError: true
};
```

Why consistency matters: Claude expects responses in a specific format. By always following the same pattern, we ensure reliable communication regardless of whether the operation succeeded or failed.

8 The Communication Protocol: JSON-RPC

8.1 Understanding JSON-RPC

JSON-RPC is like a standard language for programs to talk to each other. Here's what a typical conversation looks like:

```
// Claude sends:
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "create_page",
    "arguments": {
      "parent_id": "abc123",
      "title": "Meeting Notes",
      "content": "Discussed project timeline"
    }
  }
}

// Our server responds:
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Page created successfully with ID: def456"
      }
    ]
  }
}
```

Key components:

- **jsonrpc:** Version of the protocol
- **id:** Matches request to response (like a conversation thread)
- **method:** What action to perform
- **params:** The details of what to do
- **result:** The outcome of the operation

8.2 Why This Protocol?

Standardization: Everyone knows how to format requests and responses.

Error handling: Built-in error codes and message formats.

Request tracking: The ID system ensures responses match requests.

Language agnostic: Works the same whether you're using JavaScript, Python, or any other language.

9 Practical Setup Guide

9.1 Setting Up Your Development Environment

Step 1: Install Node.js

1. Go to nodejs.org
2. Download the LTS (Long Term Support) version
3. Run the installer with default settings
4. Verify installation: `node --version`

Step 2: Create Notion Integration

1. Go to notion.so/my-integrations
2. Click "New integration"
3. Give it a name (e.g., "Claude MCP Server")
4. Select the workspace you want to integrate with
5. Copy the "Internal Integration Token"

Step 3: Set Up Claude Desktop

1. Download Claude Desktop from claude.ai/download
2. Install and create an account
3. We'll configure MCP later

9.2 Project Structure

Create this folder organization:

```
notion-mcp-server/
          index.js              # Main MCP server
          package.json          # Project dependencies
          .env                  # Environment variables (secrets)
          .env.example          # Template for environment variables
          README.md             # Project documentation
          node_modules/         # Installed packages (auto-created)
```

9.3 Environment Variables Setup

Create a `.env` file:

```
NOTION_API_KEY=secret_abc123def456...
```

Security note: Never commit the .env file to version control. Add it to your .gitignore file.

9.4 Installation Steps

1. **Create project directory:**

```
mkdir notion-mcp-server
cd notion-mcp-server
```

2. **Install dependencies:**

```
npm install
```

3. **Create environment file:**

```
cp .env.example .env
# Edit .env and add your Notion API key
```

4. **Grant access to pages/databases:** For each page or database you want to access:
   (a) Open the page/database in Notion
   (b) Click the "..." menu (top right)
   (c) Select "Add connections"
   (d) Find and select your integration

5. **Configure Claude Desktop:** Edit your Claude Desktop configuration file:
   macOS: `~/Library/Application Support/Claude/claude_desktop_config.json`
   Windows: `%APPDATA%\Claude\claude_desktop_config.json`

```
{
  "mcpServers": {
    "notion": {
      "command": "node",
      "args": ["/absolute/path/to/your/notion-mcp-server/index.js"],
      "env": {
        "NOTION_API_KEY": "your_notion_api_key_here"
      }
    }
  }
}
```

10 Testing and Debugging

10.1 Manual Server Testing

Before connecting to Claude, test your server manually:

```
# Start the server
node index.js

# In another terminal, send a test message
echo '{"jsonrpc":"2.0","id":1,"method":"tools/list","params":{}}' | node
    index.js
```

10.2 Common Issues and Solutions

Problem: "Module not found" error
Solution: Run `npm install @notionhq/client @modelcontextprotocol/sdk zod`
Problem: "Unauthorized" error
Solution: Check your Notion API key and ensure you've granted access to the pages/databases
Problem: Claude can't connect to server
Solution: Verify the file paths in your Claude Desktop config are absolute paths
Problem: "Page not found" error
Solution: Ensure the page/database ID is correct and that your integration has been added to that page/database

10.3 Debugging Techniques

```
// Add logging to understand what's happening
console.error('Received request:', request.method);
console.error('Processing args:', args);
console.error('Notion response:', response);

// Use try-catch to isolate problems
try {
  const result = await problematicFunction();
  console.error('Success:', result);
} catch (error) {
  console.error('Failed:', error.message);
}
```

11 Available Tools and Usage

11.1 Core Tools

search_pages Search for pages in your Notion workspace.
Parameters:

- `query` (optional): Search query string
- `page_size` (optional): Number of results (default: 10, max: 100)

Example usage: "Search for pages about project planning"
create_page Create a new page in Notion.
Parameters:

- `parent_id` (required): Parent page or database ID
- `title` (required): Page title
- `content` (optional): Page content

Example usage: "Create a page called 'Meeting Notes' in my workspace"
read_page Read a specific page from Notion.
Parameters:

- `page_id` (required): Page ID to read

Example usage: "Read the content of page abc123"
update_page Update an existing page in Notion.
Parameters:

- `page_id` (required): Page ID to update
- `title` (optional): New page title
- `content` (optional): New page content

Example usage: "Update the title of page abc123 to 'Updated Meeting Notes'"
create_database Create a new database in Notion.
Parameters:

- `parent_id` (required): Parent page ID
- `title` (required): Database title
- `properties` (required): Database properties schema

Example usage: "Create a database for tracking tasks"
query_database Query a Notion database.
Parameters:

- `database_id` (required): Database ID to query
- `filter` (optional): Filter criteria
- `sorts` (optional): Sort criteria
- `page_size` (optional): Number of results (default: 10, max: 100)

Example usage: "Query my tasks database for incomplete items"

11.2 Real-World Usage Examples

Once configured, you can ask Claude to:

- "Search for pages about project planning"
- "Create a new page called 'Meeting Notes' in my workspace"
- "Read the content of page [page-id]"
- "Update the title of page [page-id] to 'New Title'"
- "Create a database for tracking tasks with properties for title, status, and due date"
- "Query my tasks database for items with status 'In Progress'"

12 Extending the System

12.1 Adding New Tools

To add a new tool, follow this pattern:

```
// 1. Add to tools list
{
  name: "get_recent_pages",
  description: "Get recently modified pages",
  inputSchema: {
    type: "object",
    properties: {
      days: {
        type: "integer",
        description: "Number of days to look back",
        default: 7
      }
    }
  }
}

// 2. Add to tool call handler
case "get_recent_pages":
  result = await getRecentPages(args);
  break;

// 3. Implement the function
async function getRecentPages(args) {
  const { days = 7 } = args;
  const cutoffDate = new Date();
  cutoffDate.setDate(cutoffDate.getDate() - days);

  const response = await notion.search({
    filter: {
      timestamp: "last_edited_time",
      last_edited_time: {
        after: cutoffDate.toISOString()
      }
    },
    sort: {
      timestamp: "last_edited_time",
      direction: "descending"
    }
  });

  return JSON.stringify(response, null, 2);
}
```

12.2 Advanced Notion Operations

```
// Working with database properties
const databaseProperties = {
```

```
  "Task Name": {
    title: {}
  },
  "Status": {
    select: {
      options: [
        { name: "Not Started", color: "red" },
        { name: "In Progress", color: "yellow" },
        { name: "Complete", color: "green" }
      ]
    }
  },
  "Due Date": {
    date: {}
  },
  "Priority": {
    number: {
      format: "number"
    }
  }
};

// Complex database queries
const response = await notion.databases.query({
  database_id: databaseId,
  filter: {
    and: [
      {
        property: "Status",
        select: {
          does_not_equal: "Complete"
        }
      },
      {
        property: "Due Date",
        date: {
          before: "2025-08-01"
        }
      }
    ]
  },
  sorts: [
    {
      property: "Priority",
      direction: "descending"
    }
  ]
});
```

12.3 Common Student Misconceptions

Misconception: "Claude is directly connected to Notion"

Reality: Claude only knows about the tools we define; the MCP server translates between

Claude and the Notion API

Misconception: "Notion automatically understands natural language"

Reality: Claude converts natural language to structured tool calls; Notion only sees API requests

Misconception: "This only works with Claude"

Reality: MCP is a standard protocol; other AI systems could use the same server

13 Conclusion

This Notion MCP Server system demonstrates several important concepts in modern software development:

13.1 Key Technical Concepts

- **API Integration:** Connecting different systems through standardized interfaces
- **Protocol Implementation:** Following MCP standards for reliable AI communication
- **Error Handling:** Building robust systems that fail gracefully
- **Asynchronous Programming:** Handling time-consuming operations efficiently
- **Data Validation:** Ensuring data integrity at system boundaries

13.2 Broader Implications

This system represents a new paradigm in human-computer interaction. Instead of learning complex user interfaces, users can accomplish tasks through natural conversation. The implications extend beyond Notion management to any domain where structured data needs to be created, queried, or modified.

Key takeaway: The technical complexity is hidden behind a simple, conversational interface. This is the power of good software design - making complex operations feel simple and natural.

13.3 Next Steps

For beginners: Focus on understanding the request-response cycle and how data flows through the system.

For intermediate learners: Experiment with adding new tools and database properties to support additional functionality.

For advanced learners: Explore integration with other systems, advanced query optimization, and scaling considerations.

Remember: The goal isn't just to build this specific system, but to understand the patterns and principles that apply to many other AI-powered applications you might create or encounter.

14 Appendix A: Complete Code Reference

14.1 Main Server File (index.js)

```
#!/usr/bin/env node

import { Server } from "@modelcontextprotocol/sdk/server/index.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/
   stdio.js";
import {
  CallToolRequestSchema,
  ListToolsRequestSchema,
  ListResourcesRequestSchema,
  ReadResourceRequestSchema,
} from "@modelcontextprotocol/sdk/types.js";
import { Client } from "@notionhq/client";
import { z } from "zod";

const notion = new Client({
  auth: process.env.NOTION_API_KEY,
});

const server = new Server(
  {
    name: "notion-mcp-server",
    version: "1.0.0",
  },
  {
    capabilities: {
      tools: {},
      resources: {},
    },
  }
);

// [Include all the schema definitions and tool handlers from the original
    code]

// Start the server
async function main() {
  const transport = new StdioServerTransport();
  await server.connect(transport);
  console.error("Notion MCP Server running on stdio");
}

main().catch((error) => {
  console.error("Server error:", error);
  process.exit(1);
});
```

14.2 Environment Variables Template

```
# Notion API Configuration
NOTION_API_KEY=your_notion_integration_token_here

# Optional: Development settings
```

```
NODE_ENV=development
DEBUG=true
```

14.3 Claude Desktop Configuration

```
{
  "mcpServers": {
    "notion": {
      "command": "node",
      "args": ["/absolute/path/to/your/notion-mcp-server/index.js"],
      "env": {
        "NOTION_API_KEY": "your_notion_api_key_here"
      }
    }
  }
}
```

14.4 Package.json Template

```
{
  "name": "notion-mcp-server",
  "version": "1.0.0",
  "description": "MCP server for Notion integration",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js"
  },
  "dependencies": {
    "@modelcontextprotocol/sdk": "^0.4.0",
    "@notionhq/client": "^2.2.15",
    "zod": "^3.22.4"
  },
  "keywords": ["mcp", "notion", "ai", "integration"],
  "author": "Your Name",
  "license": "MIT"
}
```