# 

# Server Concepts

Understanding MCP server concepts

MCP servers are programs that expose specific capabilities to AI applications through standardized protocol interfaces. Each server provides focused functionality for a particular domain.

Common examples include file system servers for document management, email servers for message handling, travel servers for trip planning, and database servers for data queries. Each server brings domain-specific capabilities to the AI application.

## Core Building Blocks

Servers provide functionality through three building blocks:

| **Building Block** | **Purpose** | **Who Controls It** | **Real-World Example** |
| --- | --- | --- | --- |
| **Tools** | For AI actions | Model-controlled | Search flights, send messages, create calendar events |
| **Resources** | For context data | Application-controlled | Documents, calendars, emails, weather data |
| **Prompts** | For interaction templates | User-controlled | "Plan a vacation", "Summarize my meetings", "Draft an email" |

### Tools - AI Actions

Tools enable AI models to perform actions through server-implemented functions. Each tool defines a specific operation with typed inputs and outputs. The model requests tool execution based on context.

#### Overview

Tools are schema-defined interfaces that LLMs can invoke. MCP uses JSON Schema for validation. Each tool performs a single operation with clearly defined inputs and outputs. Most importantly, tool execution requires explicit user approval, ensuring users maintain control over actions taken by a model.

**Protocol operations:**

| **Method** | **Purpose** | **Returns** |
| --- | --- | --- |
| tools/list | Discover available tools | Array of tool definitions with schemas |
| tools/call | Execute a specific tool | Tool execution result |

**Example tool definition:**

{

name: "searchFlights",

description: "Search for available flights",

inputSchema: {

type: "object",

properties: {

origin: { type: "string", description: "Departure city" },

destination: { type: "string", description: "Arrival city" },

date: { type: "string", format: "date", description: "Travel date" }

},

required: ["origin", "destination", "date"]

}

}

#### Example: Taking Action

Tools enable AI applications to perform actions on behalf of users. In a travel planning scenario, the AI application might use several tools to help book a vacation.

First, it searches for flights using

searchFlights(origin: "NYC", destination: "Barcelona", date: "2024-06-15")

searchFlights queries multiple airlines and returns structured flight options. Once flights are selected, it creates a calendar event with

createCalendarEvent(title: "Barcelona Trip", startDate: "2024-06-15", endDate: "2024-06-22")

to mark the travel dates. Finally, it sends an out-of-office notification using

sendEmail(to: "team@work.com", subject: "Out of Office", body: "...")

to inform colleagues about the absence.

Each tool execution requires explicit user approval, ensuring full control over actions taken.

#### User Interaction Model

Tools are model-controlled, meaning AI models can discover and invoke them automatically. However, MCP emphasizes human oversight through several mechanisms. Applications should clearly display available tools in the UI and provide visual indicators when tools are being considered or used. Before any tool execution, users must be presented with clear approval dialogs that explain exactly what the tool will do.

For trust and safety, applications often enforce manual approval to give humans the ability to deny tool invocations. Applications typically implement this through approval dialogs, permission settings for pre-approving certain safe operations, and activity logs that show all tool executions with their results.

### Resources - Context Data

Resources provide structured access to information that the host application can retrieve and provide to AI models as context.

#### Overview

Resources expose data from files, APIs, databases, or any other source that an AI needs to understand context. Applications can access this information directly and decide how to use it - whether that's selecting relevant portions, searching with embeddings, or passing it all to the model.

Resources use URI-based identification, with each resource having a unique URI such as file:///path/to/document.md. They declare MIME types for appropriate content handling and support two discovery patterns: **direct resources** with fixed URIs, and **resource templates** with parameterized URIs.

**Resource Templates** enable dynamic resource access through URI templates. A template like travel://activities/{city}/{category} would access filtered activity data by substituting both {city} and {category} parameters. For example, travel://activities/barcelona/museums would return all museums in Barcelona. Resource Templates include metadata such as title, description, and expected MIME type, making them discoverable and self-documenting.

**Protocol operations:**

| **Method** | **Purpose** | **Returns** |
| --- | --- | --- |
| resources/list | List available direct resources | Array of resource descriptors |
| resources/templates/list | Discover resource templates | Array of resource template definitions |
| resources/read | Retrieve resource contents | Resource data with metadata |
| resources/subscribe | Monitor resource changes | Subscription confirmation |

#### Example: Accessing Context Data

Continuing with the travel planning example, resources provide the AI application with access to relevant information:

* **Calendar data** (calendar://events/2024) - To check availability
* **Travel documents** (file:///Documents/Travel/passport.pdf) - For important information
* **Previous itineraries** (trips://history/barcelona-2023) - User selects which past trip style to follow

Instead of manually copying this information, resources provide raw information to AI applications. The application can choose how to best handle the data. Applications might choose to select a subset of data, using embeddings or keyword search, or pass the raw data from a resource directly to a model. In our example, during the planning phase, the AI application can pass the calendar data, weather data and travel preferences, so that the model can check availability, look up weather patterns, and reference travel preferences.

**Resource Template Examples:**

{

"uriTemplate": "weather://forecast/{city}/{date}",

"name": "weather-forecast",

"title": "Weather Forecast",

"description": "Get weather forecast for any city and date",

"mimeType": "application/json"

}

{

"uriTemplate": "travel://flights/{origin}/{destination}",

"name": "flight-search",

"title": "Flight Search",

"description": "Search available flights between cities",

"mimeType": "application/json"

}

These templates enable flexible queries. For weather data, users can access forecasts for any city/date combination. For flights, they can search routes between any two airports. When a user has input "NYC" as the origin airport and begins to input "Bar" as the destination airport, the system can suggest "Barcelona (BCN)" or "Barbados (BGI)".

#### Parameter Completion

Dynamic resources support parameter completion. For example:

* Typing "Par" as input for weather://forecast/{city} might suggest "Paris" or "Park City"
* The system helps discover valid values without requiring exact format knowledge

#### User Interaction Model

Resources are application-driven, giving hosts flexibility in how they retrieve, process, and present available context. Common interaction patterns include tree or list views for browsing resources in familiar folder-like structures, search and filter interfaces for finding specific resources, automatic context inclusion based on heuristics or AI selection, and manual selection interfaces.

Applications are free to implement resource discovery through any interface pattern that suits their needs. The protocol doesn't mandate specific UI patterns, allowing for resource pickers with preview capabilities, smart suggestions based on current conversation context, bulk selection for including multiple resources, or integration with existing file browsers and data explorers.

### Prompts - Interaction Templates

Prompts provide reusable templates. They allow MCP server authors to provide parameterized prompts for a domain, or showcase how to best use the MCP server.

#### Overview

Prompts are structured templates that define expected inputs and interaction patterns. They are user-controlled, requiring explicit invocation rather than automatic triggering. Prompts can be context-aware, referencing available resources and tools to create comprehensive workflows. Like resources, prompts support parameter completion to help users discover valid argument values.

**Protocol operations:**

| **Method** | **Purpose** | **Returns** |
| --- | --- | --- |
| prompts/list | Discover available prompts | Array of prompt descriptors |
| prompts/get | Retrieve prompt details | Full prompt definition with arguments |

#### Example: Streamlined Workflows

Prompts provide structured templates for common tasks. In the travel planning context:

**"Plan a vacation" prompt:**

{

"name": "plan-vacation",

"title": "Plan a vacation",

"description": "Guide through vacation planning process",

"arguments": [

{ "name": "destination", "type": "string", "required": true },

{ "name": "duration", "type": "number", "description": "days" },

{ "name": "budget", "type": "number", "required": false },

{ "name": "interests", "type": "array", "items": { "type": "string" } }

]

}

Rather than unstructured natural language input, the prompt system enables:

1. Selection of the "Plan a vacation" template
2. Structured input: Barcelona, 7 days, $3000, ["beaches", "architecture", "food"]
3. Consistent workflow execution based on the template

#### User Interaction Model

Prompts are user-controlled, requiring explicit invocation. Applications typically expose prompts through various UI patterns such as slash commands (typing "/" to see available prompts like /plan-vacation), command palettes for searchable access, dedicated UI buttons for frequently used prompts, or context menus that suggest relevant prompts.

The protocol gives implementers freedom to design interfaces that feel natural within their application. Key principles include easy discovery of available prompts, clear descriptions of what each prompt does, natural argument input with validation, and transparent display of the prompt's underlying template.

## How It All Works Together

The real power of MCP emerges when multiple servers work together, combining their specialized capabilities through a unified interface.

### Example: Multi-Server Travel Planning

Consider an AI application with three connected servers:

1. **Travel Server** - Handles flights, hotels, and itineraries
2. **Weather Server** - Provides climate data and forecasts
3. **Calendar/Email Server** - Manages schedules and communications

#### The Complete Flow

1. **User invokes a prompt with parameters:**

{

"prompt": "plan-vacation",

"arguments": {

"destination": "Barcelona",

"departure\_date": "2024-06-15",

"return\_date": "2024-06-22",

"budget": 3000,

"travelers": 2

}

}

1. **User selects resources to include:**
   * calendar://my-calendar/June-2024 (from Calendar Server)
   * travel://preferences/europe (from Travel Server)
   * travel://past-trips/Spain-2023 (from Travel Server)
2. **AI processes the request:**

The AI first reads all selected resources to gather context. From the calendar, it identifies available dates. From travel preferences, it learns preferred airlines and hotel types. From past trips, it discovers previously enjoyed locations. From weather data, it checks climate conditions for the travel period.

Using this context, the AI then requests user approval to execute a series of coordinated actions: searching for flights from NYC to Barcelona, finding hotels within the specified budget, creating a calendar event for the trip duration, and sending confirmation emails with the trip details.