Concepts

Prompts

Create reusable prompt templates and workflows

Prompts enable servers to define reusable prompt templates and workflows that clients can easily surface to users and LLMs. They provide a powerful way to standardize and share common LLM interactions.

> Prompts are designed to be **user-controlled**, meaning they are exposed from servers to clients with the intention of the user being able to explicitly select them for use.

Overview

Prompts in MCP are predefined templates that can:

- Accept dynamic arguments
- Include context from resources
- Chain multiple interactions
- Guide specific workflows
- Surface as UI elements (like slash commands)

Prompt structure

Each prompt is defined with:

```
{
  name: string;              // Unique identifier for the prompt
  description?: string;      // Human-readable description
```

```
  arguments?: [                // Optional list of arguments
    {
      name: string;            // Argument identifier
      description?: string;    // Argument description
      required?: boolean;      // Whether argument is required
    }
  ]
}
```

Discovering prompts

Clients can discover available prompts through the `prompts/list` endpoint:

```
// Request
{
  method: "prompts/list"
}

// Response
{
  prompts: [
    {
      name: "analyze-code",
      description: "Analyze code for potential improvements",
      arguments: [
        {
          name: "language",
          description: "Programming language",
          required: true
        }
      ]
    }
  ]
}
```

Using prompts

To use a prompt, clients make a `prompts/get` request:

```
// Request
{
  method: "prompts/get",
  params: {
    name: "analyze-code",
    arguments: {
      language: "python"
    }
  }
}

// Response
{
  description: "Analyze Python code for potential improvements",
  messages: [
    {
      role: "user",
      content: {
        type: "text",
        text: "Please analyze the following Python code for potent
      }
    }
  ]
}
```

Dynamic prompts

Prompts can be dynamic and include:

Embedded resource context

```
{
  "name": "analyze-project",
  "description": "Analyze project logs and code",
  "arguments": [
    {
```

```
        "name": "timeframe",
        "description": "Time period to analyze logs",
        "required": true
      },
      {
        "name": "fileUri",
        "description": "URI of code file to review",
        "required": true
      }
    ]
  }
```

When handling the `prompts/get` request:

```
{
  "messages": [
    {
      "role": "user",
      "content": {
        "type": "text",
        "text": "Analyze these system logs and the code file for ai
      }
    },
    {
      "role": "user",
      "content": {
        "type": "resource",
        "resource": {
          "uri": "logs://recent?timeframe=1h",
          "text": "[2024-03-14 15:32:11] ERROR: Connection timeout
          "mimeType": "text/plain"
        }
      }
    },
    {
      "role": "user",
      "content": {
        "type": "resource",
        "resource": {
          "uri": "file:///path/to/code.py",
          "text": "def connect_to_service(timeout=30):\n    retrie
```

```
            "mimeType": "text/x-python"
          }
        }
      }
    ]
  }
```

Multi-step workflows

```
const debugWorkflow = {
  name: "debug-error",
  async getMessages(error: string) {
    return [
      {
        role: "user",
        content: {
          type: "text",
          text: `Here's an error I'm seeing: ${error}`
        }
      },
      {
        role: "assistant",
        content: {
          type: "text",
          text: "I'll help analyze this error. What have you tried
        }
      },
      {
        role: "user",
        content: {
          type: "text",
          text: "I've tried restarting the service, but the error
        }
      }
    ];
  }
};
```

Example implementation

Here's a complete example of implementing prompts in an MCP server:

TypeScript Python

```typescript
import { Server } from "@modelcontextprotocol/sdk/server";
import {
  ListPromptsRequestSchema,
  GetPromptRequestSchema
} from "@modelcontextprotocol/sdk/types";

const PROMPTS = {
  "git-commit": {
    name: "git-commit",
    description: "Generate a Git commit message",
    arguments: [
      {
        name: "changes",
        description: "Git diff or description of changes",
        required: true
      }
    ]
  },
  "explain-code": {
    name: "explain-code",
    description: "Explain how code works",
    arguments: [
      {
        name: "code",
        description: "Code to explain",
        required: true
      },
      {
        name: "language",
        description: "Programming language",
        required: false
      }
    ]
  }
};
```

```
const server = new Server({
  name: "example-prompts-server",
  version: "1.0.0"
}, {
  capabilities: {
    prompts: {}
  }
});

// List available prompts
server.setRequestHandler(ListPromptsRequestSchema, async () => {
  return {
    prompts: Object.values(PROMPTS)
  };
});

// Get specific prompt
server.setRequestHandler(GetPromptRequestSchema, async (request) =>
  const prompt = PROMPTS[request.params.name];
  if (!prompt) {
    throw new Error(`Prompt not found: ${request.params.name}`);
  }

  if (request.params.name === "git-commit") {
    return {
      messages: [
        {
          role: "user",
          content: {
            type: "text",
            text: `Generate a concise but descriptive commit messa
          }
        }
      ]
    };
  }

  if (request.params.name === "explain-code") {
    const language = request.params.arguments?.language || "Unknow
    return {
      messages: [
        {
```

```
            role: "user",
            content: {
              type: "text",
              text: `Explain how this ${language} code works:\n\n${r
            }
          }
        ]
      };
    }

    throw new Error("Prompt implementation not found");
  });
```

Best practices

When implementing prompts:

1. Use clear, descriptive prompt names
2. Provide detailed descriptions for prompts and arguments
3. Validate all required arguments
4. Handle missing arguments gracefully
5. Consider versioning for prompt templates
6. Cache dynamic content when appropriate
7. Implement error handling
8. Document expected argument formats
9. Consider prompt composability
10. Test prompts with various inputs

UI integration

Prompts can be surfaced in client UIs as:

- Slash commands

Quick actions

Context menu items

Command palette entries

Guided workflows

Interactive forms

Updates and changes

Servers can notify clients about prompt changes:

1. Server capability: `prompts.listChanged`
2. Notification: `notifications/prompts/list_changed`
3. Client re-fetches prompt list

Security considerations

When implementing prompts:

Validate all arguments

Sanitize user input

Consider rate limiting

Implement access controls

Audit prompt usage

Handle sensitive data appropriately

Validate generated content

Implement timeouts

Consider prompt injection risks

Document security requirements