Concepts

Resources

Expose data and content from your servers to LLMs

Resources are a core primitive in the Model Context Protocol (MCP) that allow servers to expose data and content that can be read by clients and used as context for LLM interactions.

Resources are designed to be **application-controlled**, meaning that the client application can decide how and when they should be used. Different MCP clients may handle resources differently. For example:

- Claude Desktop currently requires users to explicitly select resources before they can be used
- Other clients might automatically select resources based on heuristics
- Some implementations may even allow the AI model itself to determine which resources to use

Server authors should be prepared to handle any of these interaction patterns when implementing resource support. In order to expose data to models automatically, server authors should use a **model-controlled** primitive such as **Tools**.

Overview

Resources represent any kind of data that an MCP server wants to make available to clients. This can include:

- File contents
- Database records
- API responses
- Live system data

Screenshots and images

Log files

And more

Each resource is identified by a unique URI and can contain either text or binary data.

Resource URIs

Resources are identified using URIs that follow this format:

```
[protocol]://[host]/[path]
```

For example:

```
file:///home/user/documents/report.pdf
postgres://database/customers/schema
screen://localhost/display1
```

The protocol and path structure is defined by the MCP server implementation. Servers can define their own custom URI schemes.

Resource types

Resources can contain two types of content:

Text resources

Text resources contain UTF-8 encoded text data. These are suitable for:

Source code

Configuration files

Log files

JSON/XML data

Binary resources

Binary resources contain raw binary data encoded in base64. These are suitable for:

- Images
- PDFs
- Audio files
- Video files
- Other non-text formats

Resource discovery

Clients can discover available resources through two main methods:

Direct resources

Servers expose a list of concrete resources via the `resources/list` endpoint. Each resource includes:

```
{
  uri: string;           // Unique identifier for the resource
  name: string;          // Human-readable name
  description?: string;  // Optional description
  mimeType?: string;     // Optional MIME type
}
```

Resource templates

For dynamic resources, servers can expose **URI templates** that clients can use to construct valid resource URIs:

```
  uriTemplate: string;    // URI template following RFC 6570
  name: string;           // Human-readable name for this type
  description?: string;   // Optional description
  mimeType?: string;      // Optional MIME type for all matching re
}
```

Reading resources

To read a resource, clients make a `resources/read` request with the resource URI.

The server responds with a list of resource contents:

```
{
  contents: [
    {
      uri: string;         // The URI of the resource
      mimeType?: string;   // Optional MIME type

      // One of:
      text?: string;       // For text resources
      blob?: string;       // For binary resources (base64 encoded)
    }
  ]
}
```

Servers may return multiple resources in response to one `resources/read` request. This could be used, for example, to return a list of files inside a directory when the directory is read.

Resource updates

MCP supports real-time updates for resources through two mechanisms:

List changes

Servers can notify clients when their list of available resources changes via the `notifications/resources/list_changed` notification.

Content changes

Clients can subscribe to updates for specific resources:

1. Client sends `resources/subscribe` with resource URI
2. Server sends `notifications/resources/updated` when the resource changes
3. Client can fetch latest content with `resources/read`
4. Client can unsubscribe with `resources/unsubscribe`

Example implementation

Here's a simple example of implementing resource support in an MCP server:

TypeScript Python

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

// List available resources
server.setRequestHandler(ListResourcesRequestSchema, async () => {
  return {
    resources: [
      {
        uri: "file:///logs/app.log",
```

```
        name: "Application Logs",
        mimeType: "text/plain"
      }
    ]
  };
});

// Read resource contents
server.setRequestHandler(ReadResourceRequestSchema, async (request
  const uri = request.params.uri;

  if (uri === "file:///logs/app.log") {
    const logContents = await readLogFile();
    return {
      contents: [
        {
          uri,
          mimeType: "text/plain",
          text: logContents
        }
      ]
    };
  }

  throw new Error("Resource not found");
});
```

Best practices

When implementing resource support:

1. Use clear, descriptive resource names and URIs
2. Include helpful descriptions to guide LLM understanding
3. Set appropriate MIME types when known
4. Implement resource templates for dynamic content
5. Use subscriptions for frequently changing resources
6. Handle errors gracefully with clear error messages

7. Consider pagination for large resource lists
8. Cache resource contents when appropriate
9. Validate URIs before processing
10. Document your custom URI schemes

Security considerations

When exposing resources:

- Validate all resource URIs
- Implement appropriate access controls
- Sanitize file paths to prevent directory traversal
- Be cautious with binary data handling
- Consider rate limiting for resource reads
- Audit resource access
- Encrypt sensitive data in transit
- Validate MIME types
- Implement timeouts for long-running reads
- Handle resource cleanup appropriately