Gmail, Calendar, and Maps MCP Server

Complete Explanation & Documentation

A comprehensive guide to understanding, setting up, and using the MCP server that integrates Gmail, Google Calendar, and Google Maps functionality.

Table of Contents

Gmail, Calendar, and Maps MCP Server - Complete Explanation

Table of Contents

Introduction

This document provides a comprehensive explanation of the **Gmail, Calendar, and Maps MCP Server** - a powerful integration server that combines three major Google services into a single, AI-accessible interface.

What This Server Does

The MCP server acts as a bridge between AI assistants (like Claude) and Google's most popular services:

- ■ **Gmail**: Read, search, and send emails
- ■ **Google Calendar**: Manage events and schedules
- ■■ **Google Maps**: Get directions, find places, and geocode addresses

Key Benefits

- **Unified Interface**: Single server handles multiple Google services

- **AI-Ready**: Designed specifically for AI assistant integration

- **Secure**: OAuth 2.0 authentication and secure API key management

- **Extensible**: Easy to add new tools and services

- **Production-Ready**: Comprehensive error handling and logging

---

What is MCP?

Model Context Protocol (MCP) is a standardized protocol that enables AI assistants to interact with external tools and services. Think of it as a "language" that AI assistants use to communicate with external systems.

How MCP Works

1. **AI Assistant** wants to perform an action (e.g., "Check my emails")

2. **MCP Client** translates the request into a tool call

3. **MCP Server** receives the tool call and executes the action

4. **External Service** (Gmail) performs the actual operation

5. **Results** flow back through the chain to the AI assistant

Why MCP Matters

- **Standardization**: Consistent interface across different services

- **Security**: Controlled access to external systems

- **Flexibility**: Easy to add new capabilities

- **Reliability**: Robust error handling and recovery

Architecture Overview

■■■■■■■■■■■■■■■■■■■■ ■■■■■■■■■■■■■■■■■■■■ ■■■■■■■■■■■■■■■■■■■■

■ AI Assistant ■ ■ MCP Client ■ ■ MCP Server ■

■ (Claude) ■■■■■■ (Claude) ■■■■■■ (Our Server) ■

■■■■■■■■■■■■■■■■■■■■ ■■■■■■■■■■■■■■■■■■■■ ■■■■■■■■■■■■■■■■■■■■

■

▼

■■■■■■■■■■■■■■■■■■■■■■■■■■■■■

■ Google APIs ■

■ ■■■■■■■ ■■■■■■■ ■■■■■■■■

■ ■Gmail■ ■Cal. ■ ■Maps ■■

■ ■■■■■■■ ■■■■■■■ ■■■■■■■■

■■■■■■■■■■■■■■■■■■■■■■■■■■■

Data Flow

1. **Request**: AI assistant makes a request
2. **Translation**: MCP client converts to tool call
3. **Execution**: MCP server calls appropriate Google API
4. **Response**: Results formatted and returned
5. **Display**: AI assistant presents results to user

Core Components

1. Main Server (`server.py`)

The heart of the system that implements the MCP protocol and manages all Google API interactions.

Key Classes

GmailCalendarMapsServer

class GmailCalendarMapsServer:

def __init__(self):

self.gmail_service = None

self.calendar_service = None

self.gmaps_client = None

self.credentials = None

This class manages:

- Google API service connections
- Authentication state
- API request handling
- Error management

Key Methods

API Initialization

async def initialize_google_apis(self, credentials_path: str, api_key: str):

```
"""Initialize Google API services"""
# Load OAuth credentials
# Build API service objects
# Set up Maps client
```

Gmail Operations

```
async def list_gmail_messages(self, query: str = "", max_results: int = 10):
"""List Gmail messages with optional search"""

async def send_gmail_message(self, to: str, subject: str, body: str):
"""Send Gmail message"""
```

Calendar Operations

```
async def list_calendar_events(self, calendar_id: str = 'primary', max_results: int = 10):
"""List calendar events"""

async def create_calendar_event(self, summary: str, start_time: str, end_time: str, ...):
"""Create calendar event"""
```

Maps Operations

```
async def get_directions(self, origin: str, destination: str, mode: str = 'driving'):
"""Get directions between locations"""

async def geocode_address(self, address: str):
"""Convert address to coordinates"""

async def find_nearby_places(self, location: str, radius: int = 5000, place_type: str = None):
"""Find nearby places"""
```

2. MCP Tools

Each tool is decorated with @mcp.server.tool() and follows a standard pattern:

```
@mcp.server.tool()
async def tool_name(request: CallToolRequest) -> CallToolResult:
"""Tool description"""
try:
# Extract arguments
args = request.arguments

# Call server method
result = await server.method_name(**args)

# Format response
return CallToolResult(
content=[TextContent(type="text", text=formatted_result)]
)
except Exception as e:
return CallToolResult(
content=[TextContent(type="text", text=f"Error: {str(e)}")]
)
```

3. Configuration Files

requirements.txt

```
mcp>=1.0.0
google-auth>=2.23.0
google-auth-oauthlib>=1.1.0
googleapiclient.discovery>=2.100.0
googlemaps>=4.10.0
```

... other dependencies

mcp-config.json

```
{
"mcpServers": {
"gmail-calendar-maps": {
"command": "python",
"args": ["server.py"],
"env": {
"GOOGLE_MAPS_API_KEY": "YOUR_GOOGLE_MAPS_API_KEY"
}
}
}
}
```

Authentication & Security

OAuth 2.0 Flow

1. **Setup Phase**

User → Google Cloud Console → Download client_secrets.json

2. **Authentication Phase**

User → setup_google_apis.py → Browser opens → Google OAuth → credentials.json

3. **Runtime Phase**

Server → Load credentials.json → Refresh if needed → Make API calls

Security Features

Credential Management

- OAuth 2.0 for Gmail and Calendar
- API keys for Maps (stored in environment variables)
- Automatic token refresh
- Secure credential storage

Access Control

- Scoped permissions (read-only, send, calendar access)
- User consent required for each scope
- Revocable access tokens

Error Handling

- Graceful failure on authentication errors
- Descriptive error messages
- No sensitive data in error logs

Required Scopes

```
SCOPES = [
'https://www.googleapis.com/auth/gmail.readonly',
'https://www.googleapis.com/auth/gmail.send',
'https://www.googleapis.com/auth/calendar',
'https://www.googleapis.com/auth/calendar.events',
]
```

Available Tools

Gmail Tools

1. `list_gmail_messages_tool`

Purpose: List recent Gmail messages with optional search filtering

Parameters:
- query (string, optional): Gmail search query
- max_results (integer, optional): Maximum number of messages

Example:
```
{
"name": "list_gmail_messages_tool",
"arguments": {
"query": "is:unread from:work.com",
"max_results": 10
}
}
```

Response:
```
Recent Gmail Messages:

From: john@work.com
Subject: Weekly Team Update
Date: Mon, 15 Jan 2024 09:30:00 +0000
Snippet: Here's this week's progress report...
--------------------------------------------------
```

2. `send_gmail_message_tool`
Purpose: Send a Gmail message

Parameters:
- to (string, required): Recipient email address
- subject (string, required): Email subject
- body (string, required): Email body content

Example:
```
{
"name": "send_gmail_message_tool",
"arguments": {
"to": "colleague@company.com",
"subject": "Meeting Reminder",
"body": "Don't forget our meeting tomorrow at 2 PM!"
}
}
```

Calendar Tools

3. `list_calendar_events_tool`

Purpose: List upcoming calendar events

Parameters:

- calendar_id (string, optional): Calendar ID (default: "primary")
- max_results (integer, optional): Maximum number of events

Example:

```
{
"name": "list_calendar_events_tool",
"arguments": {
"calendar_id": "primary",
"max_results": 5
}
}
```

4. `create_calendar_event_tool`

Purpose: Create a new calendar event

Parameters:

- summary (string, required): Event title
- start_time (string, required): Start time in ISO format
- end_time (string, required): End time in ISO format
- description (string, optional): Event description
- location (string, optional): Event location
- attendees (array, optional): List of attendee email addresses

Example:

```
{
"name": "create_calendar_event_tool",
"arguments": {
"summary": "Team Standup",
"start_time": "2024-01-16T10:00:00Z",
"end_time": "2024-01-16T10:30:00Z",
"description": "Daily team sync meeting",
"location": "Conference Room A",
"attendees": ["team@company.com"]
}
}
```

Maps Tools

5. `get_directions_tool`

Purpose: Get directions between two locations

Parameters:

- origin (string, required): Starting location
- destination (string, required): Destination location
- mode (string, optional): Travel mode ("driving", "walking", "bicycling", "transit")

Example:

```
{
"name": "get_directions_tool",
"arguments": {
"origin": "San Francisco, CA",
"destination": "Mountain View, CA",
"mode": "driving"
}
}
```

Response:

```
Directions from San Francisco, CA to Mountain View, CA:
Distance: 35.2 mi
Duration: 45 mins

Steps:
1. Head south on US-101 S (35.2 mi, 45 mins)
2. Take exit 400B for CA-85 toward Cupertino (0.5 mi, 1 min)
3. Turn right onto CA-85 S (2.1 mi, 3 mins)
4. Arrive at Mountain View, CA
```

6. `geocode_address_tool`

Purpose: Convert address to coordinates

Parameters:

- address (string, required): Address to geocode

Example:

```
{
"name": "geocode_address_tool",
"arguments": {
"address": "1600 Amphitheatre Parkway, Mountain View, CA"
}
```

}

7. `find_nearby_places_tool`
Purpose: Find nearby places around a location

Parameters:
- location (string, required): Center location
- radius (integer, optional): Search radius in meters (default: 5000)
- place_type (string, optional): Type of place to search for

Example:
{
"name": "find_nearby_places_tool",
"arguments": {
"location": "San Francisco, CA",
"radius": 2000,
"place_type": "restaurant"
}
}

Usage Examples

Email Management Workflow

1. Check unread emails

unread = await list_gmail_messages_tool({
"query": "is:unread",
"max_results": 5
})

2. Search for specific emails

work_emails = await list_gmail_messages_tool({
"query": "from:work.com OR subject:meeting",
"max_results": 10

})

3. Send a response

await send_gmail_message_tool({
"to": "sender@example.com",
"subject": "Re: Your Question",
"body": "Thanks for your email. I'll get back to you soon."
})

Calendar Management Workflow

1. Check upcoming events

events = await list_calendar_events_tool({
"max_results": 10
})

2. Create a new meeting

await create_calendar_event_tool({
"summary": "Project Review Meeting",
"start_time": "2024-01-20T14:00:00Z",
"end_time": "2024-01-20T15:00:00Z",
"description": "Quarterly project review with stakeholders",
"location": "Conference Room B",
"attendees": ["stakeholder@company.com", "manager@company.com"]
})

Location Services Workflow

1. Get directions to a meeting

```
directions = await get_directions_tool({
"origin": "Current Location",
"destination": "Meeting Location",
"mode": "driving"
})
```

2. Find lunch options near the meeting

```
restaurants = await find_nearby_places_tool({
"location": "Meeting Location",
"radius": 1000,
"place_type": "restaurant"
})
```

3. Geocode the restaurant address

```
coords = await geocode_address_tool({
"address": restaurants[0]['address']
})
```

Integration Workflows

Business Trip Planning

Here's how all three services work together to plan a business trip:

Step 1: Check calendar availability

```
events = await list_calendar_events_tool({
"max_results": 20
})
```

Step 2: Find hotels at destination

```
hotels = await find_nearby_places_tool({
"location": "San Francisco, CA",
"radius": 3000,
"place_type": "lodging"
})
```

Step 3: Get directions from airport to hotel

```
directions = await get_directions_tool({
"origin": "San Francisco International Airport",
"destination": hotels[0]['address'],
"mode": "driving"
})
```

Step 4: Create trip calendar event

```
await create_calendar_event_tool({
"summary": "Business Trip to San Francisco",
"start_time": "2024-01-25T09:00:00Z",
"end_time": "2024-01-25T17:00:00Z",
"description": f"Business trip. Hotel: {hotels[0]['name']}. Travel time from airport:
{directions['duration']}",
"location": "San Francisco, CA"
})
```

Step 5: Send trip confirmation email

```
await send_gmail_message_tool({
"to": "manager@company.com",
"subject": "Business Trip Confirmed",
"body": f"Trip to San Francisco confirmed for Jan 25. Staying at {hotels[0]['name']}."
})
```

Meeting Coordination

Step 1: Find available meeting time

```
events = await list_calendar_events_tool({
"max_results": 10
})
```

Step 2: Find meeting location

```
restaurants = await find_nearby_places_tool({
"location": "Downtown",
"radius": 2000,
"place_type": "restaurant"
})
```

Step 3: Get directions for attendees

```
directions = await get_directions_tool({
"origin": "Office",
"destination": restaurants[0]['address'],
"mode": "walking"
})
```

Step 4: Create meeting event

```
await create_calendar_event_tool({
"summary": "Lunch Meeting",
"start_time": "2024-01-18T12:00:00Z",
"end_time": "2024-01-18T13:00:00Z",
```

"description": f"Lunch at {restaurants[0]['name']}. Walking distance: {directions['distance']}",

"location": restaurants[0]['address'],

"attendees": ["colleague@company.com"]

})

Step 5: Send meeting invitation

await send_gmail_message_tool({

"to": "colleague@company.com",

"subject": "Lunch Meeting Tomorrow",

"body": f"Let's meet at {restaurants[0]['name']} tomorrow at noon. It's a {directions['distance']} walk from the office."

})

Setup & Configuration

Prerequisites

1. **Python 3.8+** installed
2. **Google Cloud Project** with APIs enabled
3. **OAuth 2.0 credentials** for Gmail and Calendar
4. **Google Maps API key** for location services

Step-by-Step Setup

1. Google Cloud Console Setup

1. Go to Google Cloud Console

2. Create a new project or select existing one

3. Enable required APIs:

- Gmail API
- Google Calendar API
- Maps JavaScript API
- Directions API
- Places API

2. Create OAuth 2.0 Credentials
1. Go to "Credentials" section
2. Click "Create Credentials" → "OAuth 2.0 Client IDs"
3. Choose "Desktop application"
4. Download JSON file and rename to client_secrets.json
5. Place in project directory

3. Get Google Maps API Key
1. In Google Cloud Console, go to "Credentials"
2. Click "Create Credentials" → "API Key"
3. Copy the API key

4. Run Setup Script

```
python setup_google_apis.py
```

This script will:
- Guide you through OAuth authentication
- Save credentials to credentials.json
- Help set up environment variables

5. Configure MCP Client
Add to your MCP client configuration:

```
{
"mcpServers": {
"gmail-calendar-maps": {
"command": "python",
"args": ["server.py"],
"env": {
"GOOGLE_MAPS_API_KEY": "YOUR_GOOGLE_MAPS_API_KEY"
}
}
}
}
```

Environment Variables

Create a .env file:

Google Maps API Key

```
GOOGLE_MAPS_API_KEY=your_google_maps_api_key_here
```

Optional: Google Cloud Project ID

```
GOOGLE_CLOUD_PROJECT_ID=your_project_id_here
```

Optional: Logging level

```
LOG_LEVEL=INFO
```

Customization & Extension

Adding New Google APIs

To add a new Google API (e.g., Google Drive):

1. **Update scopes** in server.py:

```
SCOPES = [
# ... existing scopes ...
'https://www.googleapis.com/auth/drive.readonly',
]
```

2. **Add service initialization**:

```
self.drive_service = build('drive', 'v3', credentials=self.credentials)
```

3. **Add methods**:

```
async def list_drive_files(self, query: str = "", max_results: int = 10):
"""List Google Drive files"""
# Implementation here
```

4. **Create MCP tool**:

```
@mcp.server.tool()
async def list_drive_files_tool(request: CallToolRequest) -> CallToolResult:
"""List Google Drive files"""
# Implementation here
```

Adding Custom Tools

You can add tools that don't use Google APIs:

```
@mcp.server.tool()
async def custom_calculation_tool(request: CallToolRequest) -> CallToolResult:
"""Perform custom calculations"""
try:
args = request.arguments
a = args.get("a", 0)
b = args.get("b", 0)
operation = args.get("operation", "add")

if operation == "add":
result = a + b
elif operation == "multiply":
result = a * b
else:
return CallToolResult(
content=[TextContent(type="text", text="Error: Invalid operation")]
)

return CallToolResult(
content=[TextContent(type="text", text=f"Result: {result}")]
)
except Exception as e:
return CallToolResult(
content=[TextContent(type="text", text=f"Error: {str(e)}")]
)
```

Performance Optimizations

1. **Caching**:

```
import functools
import asyncio

@functools.lru_cache(maxsize=128)
async def cached_geocode(address: str):
"""Cache geocoding results"""
return await server.geocode_address(address)
```

2. **Batch Operations**:

```
async def batch_create_events(self, events: List[Dict]):
```

"""Create multiple events efficiently"""
results = []
for event in events:
result = await self.create_calendar_event(**event)
results.append(result)
return results

3. **Connection Pooling**:

Use connection pooling for HTTP requests

import aiohttp

async with aiohttp.ClientSession() as session:
# Reuse session for multiple requests
pass

Troubleshooting

Common Issues

1. Authentication Errors
Problem: "Invalid credentials" or "Token expired"

Solutions:
- Run python setup_google_apis.py to refresh credentials
- Check that client_secrets.json is in the project directory
- Verify OAuth scopes are properly configured
- Ensure credentials haven't been revoked

2. API Key Issues
Problem: "API key not valid" or "Quota exceeded"

Solutions:
- Verify Google Maps API key is correct
- Check that required APIs are enabled in Google Cloud Console
- Ensure billing is set up for the project
- Monitor API usage in Google Cloud Console

3. Rate Limiting

Problem: "Quota exceeded" or "Rate limit exceeded"

Solutions:
- Implement exponential backoff for retries
- Cache frequently accessed data
- Monitor API usage and quotas
- Consider upgrading API quotas if needed

4. Network Issues

Problem: "Connection timeout" or "Network error"

Solutions:
- Check internet connectivity
- Verify firewall settings
- Try with different network
- Implement retry logic with backoff

Debug Mode

Enable debug logging:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Error Logs

Check logs for detailed error information:

Run server with verbose logging

```
python server.py --verbose
```

Check specific error types

```
grep "ERROR" server.log
```

Testing Individual Components

Test specific functionality:

Test Gmail only

python -c "

import asyncio

from server import GmailCalendarMapsServer

server = GmailCalendarMapsServer()

Test Gmail methods

"

Test Calendar only

python -c "

Test Calendar methods

"

Test Maps only

python -c "

Test Maps methods

"

Best Practices

Security Best Practices

1. **Credential Management**
- Never commit credentials to version control
- Use environment variables for API keys
- Rotate credentials regularly
- Use least-privilege access

2. **API Usage**
- Monitor API quotas and usage
- Implement rate limiting
- Cache responses when appropriate
- Handle errors gracefully

3. **Data Privacy**
- Minimize data collection
- Secure data transmission
- Implement access controls
- Regular security audits

Performance Best Practices

1. **Efficient API Usage**
- Batch operations when possible
- Use appropriate page sizes
- Implement caching strategies
- Monitor response times

2. **Resource Management**
- Close connections properly
- Use connection pooling
- Implement timeouts
- Monitor memory usage

3. **Error Handling**
- Implement retry logic
- Use exponential backoff
- Log errors appropriately
- Provide user-friendly messages

Code Quality

1. **Documentation**
- Document all public methods
- Include usage examples
- Maintain up-to-date README
- Add inline comments for complex logic

2. **Testing**
- Write unit tests for core functionality
- Test error conditions
- Mock external dependencies
- Maintain good test coverage

3. **Maintenance**
- Regular dependency updates
- Monitor for deprecation warnings
- Keep up with API changes
- Regular code reviews

Deployment Best Practices

1. **Environment Setup**
- Use virtual environments
- Pin dependency versions
- Use configuration management
- Implement proper logging

2. **Monitoring**
- Monitor server health
- Track API usage
- Alert on errors
- Performance metrics

3. **Backup & Recovery**
- Backup configuration files
- Document recovery procedures
- Test disaster recovery
- Version control everything

Conclusion

The Gmail, Calendar, and Maps MCP Server provides a powerful, secure, and extensible way to integrate Google's most popular services with AI assistants. By following the patterns and best practices outlined in this document, you can build robust, production-ready integrations that enhance the capabilities of AI systems.

Key Takeaways

1. **MCP Protocol**: Provides a standardized way for AI assistants to interact with external services
2. **Google APIs**: Rich ecosystem of services that can be integrated
3. **Security**: OAuth 2.0 and proper credential management are essential
4. **Extensibility**: Easy to add new tools and services
5. **Best Practices**: Follow security, performance, and code quality guidelines

Next Steps

1. **Set up the server** using the provided instructions
2. **Test the functionality** with the included test suite
3. **Customize for your needs** by adding new tools or APIs
4. **Deploy to production** following best practices
5. **Monitor and maintain** the system

Resources

- MCP Protocol Documentation
- Google API Documentation
- Gmail API Guide
- Google Calendar API Guide
- Google Maps API Guide

This document provides a comprehensive guide to understanding, setting up, and using the Gmail, Calendar, and Maps MCP Server. For questions or support, please refer to the troubleshooting section or create an issue in the project repository.