---
name: "MCP Server PRP Template"
description: This template is designed to provide a production-ready Model Context Protocol (MCP) server using the proven patterns from this codebase.

Purpose

Template optimized for AI agents to implement production-ready Model Context Protocol (MCP) servers with GitHub OAuth authentication, database integration, and Cloudflare Workers deployment using the proven patterns from this codebase.

Core Principles

1. **Context is King**: Include ALL necessary MCP patterns, authentication flows, and deployment configurations
2. **Validation Loops**: Provide executable tests from TypeScript compilation to production deployment
3. **Security First**: Build-in authentication, authorization, and SQL injection protection
4. **Production Ready**: Include monitoring, error handling, and deployment automation

---

Goal

Build a production-ready MCP (Model Context Protocol) server with:

- [SPECIFIC MCP FUNCTIONALITY] - describe the specific tools and resources to implement
- GitHub OAuth authentication with role-based access control
- Cloudflare Workers deployment with monitoring
- [ADDITIONAL FEATURES] - any specific features beyond the base authentication/database

Why

- **Developer Productivity**: Enable secure AI assistant access to [SPECIFIC DATA/OPERATIONS]
- **Enterprise Security**: GitHub OAuth with granular permission system
- **Scalability**: Cloudflare Workers global edge deployment
- **Integration**: [HOW THIS FITS WITH EXISTING SYSTEMS]
- **User Value**: [SPECIFIC BENEFITS TO END USERS]

What

MCP Server Features

Core MCP Tools:

- Tools are organized in modular files and registered via `src/tools/register-tools.ts`
- Each feature/domain gets its own tool registration file (e.g., `database-tools.ts`, `analytics-tools.ts`)
- [LIST SPECIFIC TOOLS] - e.g., "queryDatabase", "listTables", "executeOperations"
- User authentication and permission validation happens during tool registration
- Comprehensive error handling and logging
- [DOMAIN-SPECIFIC TOOLS] - tools specific to your use case

Authentication & Authorization:

- GitHub OAuth 2.0 integration with signed cookie approval system
- Role-based access control (read-only vs privileged users)
- User context propagation to all MCP tools
- Secure session management with HMAC-signed cookies

Database Integration:

- PostgreSQL connection pooling with automatic cleanup
- SQL injection protection and query validation
- Read/write operation separation based on user permissions
- Error sanitization to prevent information leakage

Deployment & Monitoring:

- Cloudflare Workers with Durable Objects for state management
- Optional Sentry integration for error tracking and performance monitoring
- Environment-based configuration (development vs production)
- Real-time logging and alerting

Success Criteria

- [ ] MCP server passes validation with MCP Inspector
- [ ] GitHub OAuth flow works end-to-end (authorization → callback → MCP access)
- [ ] TypeScript compilation succeeds with no errors
- [ ] Local development server starts and responds correctly
- [ ] Production deployment to Cloudflare Workers succeeds
- [ ] Authentication prevents unauthorized access to sensitive operations
- [ ] Error handling provides user-friendly messages without leaking system details
- [ ] [DOMAIN-SPECIFIC SUCCESS CRITERIA]

All Needed Context

Documentation & References (MUST READ)

```yaml
# CRITICAL MCP PATTERNS - Read these first
- docfile: PRPs/ai_docs/mcp_patterns.md
  why: Core MCP development patterns, security practices, and error handling

# Critial code examples
- docfile: PRPs/ai_docs/claude_api_usage.md
  why: How to use the Anthropic API to get a response from an LLM

# TOOL REGISTRATION SYSTEM - Understand the modular approach
- file: src/tools/register-tools.ts
  why: Central registry showing how all tools are imported and registered - STUDY this pattern

# EXAMPLE MCP TOOLS - Look here how to create and register new tools
- file: examples/database-tools.ts
  why: Example tools for a Postgres MCP server showing best practices for tool creation and registration

- file: examples/database-tools-sentry.ts
  why: Example tools for the Postgres MCP server but with the Sentry integration for production monitoring

# EXISTING CODEBASE PATTERNS - Study these implementations
- file: src/index.ts
  why: Complete MCP server with authentication, database, and tools - MIRROR this pattern

- file: src/github-handler.ts
  why: OAuth flow implementation - USE this exact pattern for authentication

- file: src/database.ts
  why: Database security, connection pooling, SQL validation - FOLLOW these patterns

- file: wrangler.jsonc
  why: Cloudflare Workers configuration - COPY this pattern for deployment

# OFFICIAL MCP DOCUMENTATION
- url: https://modelcontextprotocol.io/docs/concepts/tools
  why: MCP tool registration and schema definition patterns

- url: https://modelcontextprotocol.io/docs/concepts/resources
  why: MCP resource implementation if needed

# Add n documentation related to the users use case as needed below
```

Current Codebase Tree (Run `tree -I node_modules` in project root)

```bash
# INSERT ACTUAL TREE OUTPUT HERE
/
├── src/
│   ├── index.ts              # Main authenticated MCP server ← STUDY THIS
│   ├── index_sentry.ts       # Sentry monitoring version
│   ├── simple-math.ts        # Basic MCP example ← GOOD STARTING POINT
│   ├── github-handler.ts     # OAuth implementation ← USE THIS PATTERN
│   ├── database.ts           # Database utilities ← SECURITY PATTERNS
│   ├── utils.ts              # OAuth helpers
│   ├── workers-oauth-utils.ts  # Cookie security system
│   └── tools/                # Tool registration system
│       └── register-tools.ts   # Central tool registry ← UNDERSTAND THIS
├── PRPs/
│   ├── templates/prp_mcp_base.md  # This template
│   └── ai_docs/              # Implementation guides ← READ ALL
├── examples/                 # Example tool implementations
│   ├── database-tools.ts     # Database tools example ← FOLLOW PATTERN
│   └── database-tools-sentry.ts # With Sentry monitoring
├── wrangler.jsonc            # Cloudflare config ← COPY PATTERNS
├── package.json              # Dependencies
└── tsconfig.json             # TypeScript config
```

Desired Codebase Tree (Files to add/modify) related to the users use case as needed below

```bash

```

Known Gotchas & Critical MCP/Cloudflare Patterns

```typescript
// CRITICAL: Cloudflare Workers require specific patterns
// 1. ALWAYS implement cleanup for Durable Objects
export class YourMCP extends McpAgent<Env, Record<string, never>, Props> {
 async cleanup(): Promise<void> {
  await closeDb(); // CRITICAL: Close database connections
 }

 async alarm(): Promise<void> {
  await this.cleanup(); // CRITICAL: Handle Durable Object alarms
```

```
 }
}

// 2. ALWAYS validate SQL to prevent injection (use existing patterns)
const validation = validateSqlQuery(sql); // from src/database.ts
if (!validation.isValid) {
 return createErrorResponse(validation.error);
}

// 3. ALWAYS check permissions before sensitive operations
const ALLOWED_USERNAMES = new Set(["admin1", "admin2"]);
if (!ALLOWED_USERNAMES.has(this.props.login)) {
 return createErrorResponse("Insufficient permissions");
}

// 4. ALWAYS use withDatabase wrapper for connection management
return await withDatabase(this.env.DATABASE_URL, async (db) => {
 // Database operations here
});

// 5. ALWAYS use Zod for input validation
import { z } from "zod";
const schema = z.object({
 param: z.string().min(1).max(100),
});

// 6. TypeScript compilation requires exact interface matching
interface Env {
 DATABASE_URL: string;
 GITHUB_CLIENT_ID: string;
 GITHUB_CLIENT_SECRET: string;
 OAUTH_KV: KVNamespace;
 // Add your environment variables here
}
```

Implementation Blueprint

Data Models & Types

Define TypeScript interfaces and Zod schemas for type safety and validation.

```typescript
// User authentication props (inherited from OAuth)
type Props = {
 login: string; // GitHub username
 name: string; // Display name
```

```
  email: string; // Email address
  accessToken: string; // GitHub access token
};

// MCP tool input schemas (customize for your tools)
const YourToolSchema = z.object({
  param1: z.string().min(1, "Parameter cannot be empty"),
  param2: z.number().int().positive().optional(),
  options: z.object({}).optional(),
});

// Environment interface (add your variables)
interface Env {
  DATABASE_URL: string;
  GITHUB_CLIENT_ID: string;
  GITHUB_CLIENT_SECRET: string;
  OAUTH_KV: KVNamespace;
  // YOUR_SPECIFIC_ENV_VAR: string;
}

// Permission levels (customize for your use case)
enum Permission {
  READ = "read",
  WRITE = "write",
  ADMIN = "admin",
}
```

List of Tasks (Complete in order)

```yaml
Task 1 - Project Setup:
  COPY wrangler.jsonc to wrangler-[server-name].jsonc:
    - MODIFY name field to "[server-name]"
    - ADD any new environment variables to vars section
    - KEEP existing OAuth and database configuration

  CREATE .dev.vars file (if not exists):
    - ADD GITHUB_CLIENT_ID=your_client_id
    - ADD GITHUB_CLIENT_SECRET=your_client_secret
    - ADD DATABASE_URL=postgresql://...
    - ADD COOKIE_ENCRYPTION_KEY=your_32_byte_key
    - ADD any domain-specific environment variables

Task 2 - GitHub OAuth App:
  CREATE new GitHub OAuth app:
    - SET homepage URL: https://your-worker.workers.dev
```

- SET callback URL: https://your-worker.workers.dev/callback
- COPY client ID and secret to .dev.vars

OR REUSE existing OAuth app:
- UPDATE callback URL if using different subdomain
- VERIFY client ID and secret in environment

Task 3 - MCP Server Implementation:
CREATE src/[server-name].ts OR MODIFY src/index.ts:
- COPY class structure from src/index.ts
- MODIFY server name and version in McpServer constructor
- CALL registerAllTools(server, env, props) in init() method
- KEEP authentication and database patterns identical

CREATE tool modules:
- CREATE new tool files following examples/database-tools.ts pattern
- EXPORT registration functions that accept (server, env, props)
- USE Zod schemas for input validation
- IMPLEMENT proper error handling with createErrorResponse
- ADD permission checking during tool registration

UPDATE tool registry:
- MODIFY src/tools/register-tools.ts to import your new tools
- ADD your registration function call in registerAllTools()

Task 4 - Database Integration (if needed):
USE existing database patterns from src/database.ts:
- IMPORT withDatabase, validateSqlQuery, isWriteOperation
- IMPLEMENT database operations with security validation
- SEPARATE read vs write operations based on user permissions
- USE formatDatabaseError for user-friendly error messages

Task 5 - Environment Configuration:
SETUP Cloudflare KV namespace:
- RUN: wrangler kv namespace create "OAUTH_KV"
- UPDATE wrangler.jsonc with returned namespace ID

SET production secrets:
- RUN: wrangler secret put GITHUB_CLIENT_ID
- RUN: wrangler secret put GITHUB_CLIENT_SECRET
- RUN: wrangler secret put DATABASE_URL
- RUN: wrangler secret put COOKIE_ENCRYPTION_KEY

Task 6 - Local Testing:
TEST basic functionality:
- RUN: wrangler dev
- VERIFY server starts without errors

```
   - TEST OAuth flow: http://localhost:8792/authorize
   - VERIFY MCP endpoint: http://localhost:8792/mcp

Task 7 - Production Deployment:
  DEPLOY to Cloudflare Workers:
   - RUN: wrangler deploy
   - VERIFY deployment success
   - TEST production OAuth flow
   - VERIFY MCP endpoint accessibility
```

Per Task Implementation Details

```typescript
// Task 3 - MCP Server Implementation Pattern
export class YourMCP extends McpAgent<Env, Record<string, never>, Props> {
  server = new McpServer({
    name: "Your MCP Server Name",
    version: "1.0.0",
  });

  // CRITICAL: Always implement cleanup
  async cleanup(): Promise<void> {
    try {
      await closeDb();
      console.log("Database connections closed successfully");
    } catch (error) {
      console.error("Error during database cleanup:", error);
    }
  }

  async alarm(): Promise<void> {
    await this.cleanup();
  }

  async init() {
    // PATTERN: Use centralized tool registration
    registerAllTools(this.server, this.env, this.props);
  }
}

// Task 3 - Tool Module Pattern (e.g., src/tools/your-feature-tools.ts)
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { Props } from "../types";
import { z } from "zod";

const PRIVILEGED_USERS = new Set(["admin1", "admin2"]);
```

```
export function registerYourFeatureTools(server: McpServer, env: Env, props: Props) {
  // Tool 1: Available to all authenticated users
  server.tool(
    "yourBasicTool",
    "Description of your basic tool",
    YourToolSchema, // Zod validation schema
    async ({ param1, param2, options }) => {
      try {
        // PATTERN: Tool implementation with error handling
        const result = await performOperation(param1, param2, options);

        return {
          content: [
            {
              type: "text",
              text: `**Success**\n\nOperation
completed\n\n**Result:**\n\`\`\`json\n${JSON.stringify(result, null, 2)}\n\`\`\``,
            },
          ],
        };
      } catch (error) {
        return createErrorResponse(`Operation failed: ${error.message}`);
      }
    },
  );

  // Tool 2: Only for privileged users
  if (PRIVILEGED_USERS.has(props.login)) {
    server.tool(
      "privilegedTool",
      "Administrative tool for privileged users",
      { action: z.string() },
      async ({ action }) => {
        // Implementation
        return {
          content: [
            {
              type: "text",
              text: `Admin action '${action}' executed by ${props.login}`,
            },
          ],
        };
      },
    );
  }
}
```

```
// Task 3 - Update Tool Registry (src/tools/register-tools.ts)
import { registerYourFeatureTools } from "./your-feature-tools";

export function registerAllTools(server: McpServer, env: Env, props: Props) {
  // Existing registrations
  registerDatabaseTools(server, env, props);

  // Add your new registration
  registerYourFeatureTools(server, env, props);
}

// PATTERN: Export OAuth provider with MCP endpoints
export default new OAuthProvider({
  apiHandlers: {
    "/sse": YourMCP.serveSSE("/sse") as any,
    "/mcp": YourMCP.serve("/mcp") as any,
  },
  authorizeEndpoint: "/authorize",
  clientRegistrationEndpoint: "/register",
  defaultHandler: GitHubHandler as any,
  tokenEndpoint: "/token",
});
```

Integration Points

```yaml
CLOUDFLARE_WORKERS:
  - wrangler.jsonc: Update name, environment variables, KV bindings
  - Environment secrets: GitHub OAuth credentials, database URL, encryption key
  - Durable Objects: Configure MCP agent binding for state persistence

GITHUB_OAUTH:
  - GitHub App: Create with callback URL matching your Workers domain
  - Client credentials: Store as Cloudflare Workers secrets
  - Callback URL: Must match exactly: https://your-worker.workers.dev/callback

DATABASE:
  - PostgreSQL connection: Use existing connection pooling patterns
  - Environment variable: DATABASE_URL with full connection string
  - Security: Use validateSqlQuery and isWriteOperation for all SQL

ENVIRONMENT_VARIABLES:
  - Development: .dev.vars file for local testing
  - Production: Cloudflare Workers secrets for deployment
```

- Required: GITHUB_CLIENT_ID, GITHUB_CLIENT_SECRET, DATABASE_URL, COOKIE_ENCRYPTION_KEY

KV_STORAGE:
- OAuth state: Used by OAuth provider for state management
- Namespace: Create with `wrangler kv namespace create "OAUTH_KV"`
- Configuration: Add namespace ID to wrangler.jsonc bindings
```

## Validation Gate

### Level 1: TypeScript & Configuration

```bash
# CRITICAL: Run these FIRST - fix any errors before proceeding
npm run type-check              # TypeScript compilation
wrangler types                  # Generate Cloudflare Workers types

# Expected: No TypeScript errors
# If errors: Fix type issues, missing interfaces, import problems
```

Level 2: Local Development Testing

```bash
# Start local development server
wrangler dev

# Test OAuth flow (should redirect to GitHub)
curl -v http://localhost:8792/authorize

# Test MCP endpoint (should return server info)
curl -v http://localhost:8792/mcp

# Expected: Server starts, OAuth redirects to GitHub, MCP responds with server info
# If errors: Check console output, verify environment variables, fix configuration
```

Level 3: Unit test each feature, function, and file, following existing testing patterns if they are there.

```bash
npm run test
```

Run unit tests with the above command (Vitest) to make sure all functionality is working.
```

Level 4: Database Integration Testing (if applicable)

```bash
# Test database connection
curl -X POST http://localhost:8792/mcp \
  -H "Content-Type: application/json" \
  -d '{"method": "tools/call", "params": {"name": "listTables", "arguments": {}}}'

# Test permission validation
# Test SQL injection protection and other kinds of security if applicable
# Test error handling for database failures

# Expected: Database operations work, permissions enforced, errors handled gracefully, etc.
# If errors: Check DATABASE_URL, connection settings, permission logic
```

Final Validation Checklist

Core Functionality

- [ ] TypeScript compilation: `npm run type-check` passes
- [ ] Unit tests pass: `npm run test` passes
- [ ] Local server starts: `wrangler dev` runs without errors
- [ ] MCP endpoint responds: `curl http://localhost:8792/mcp` returns server info
- [ ] OAuth flow works: Authentication redirects and completes successfully

---

Anti-Patterns to Avoid

MCP-Specific

- ❌ Don't skip input validation with Zod - always validate tool parameters
- ❌ Don't forget to implement cleanup() method for Durable Objects
- ❌ Don't hardcode user permissions - use configurable permission systems

Development Process

- ❌ Don't skip the validation loops - each level catches different issues
- ❌ Don't guess about OAuth configuration - test the full flow
- ❌ Don't deploy without monitoring - implement logging and error tracking
- ❌ Don't ignore TypeScript errors - fix all type issues before deployment