# SSH Integration Implementation Plan

## Overview

This document outlines the implementation plan for integrating SSH connectivity into the application. This feature will allow users to:

1. Store SSH connection details for remote servers
2. Test SSH connections directly from the application
3. Install and manage MCP server instances on remote machines
4. Execute commands on remote servers through a secure SSH tunnel

## Data Model

### SSH Connection Table

```
CREATE TABLE ssh_connections (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  user_id UUID NOT NULL,
  name VARCHAR(100) NOT NULL,
  host VARCHAR(255) NOT NULL,
  port INTEGER NOT NULL DEFAULT 22,
  username VARCHAR(100) NOT NULL,
  auth_type VARCHAR(20) NOT NULL DEFAULT 'password', -- 'password', 'key', 'agent'
  password_encrypted TEXT,
  private_key_encrypted TEXT,
  passphrase_encrypted TEXT,
  fingerprint VARCHAR(255),
  last_connected TIMESTAMP,
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE,
  CONSTRAINT unique_connection_per_user UNIQUE (user_id, name)
);

-- Index for faster lookups
CREATE INDEX idx_ssh_connections_user_id ON ssh_connections(user_id);
CREATE INDEX idx_ssh_connections_is_active ON ssh_connections(is_active);
```

### SSH Connection History Table

```
CREATE TABLE ssh_connection_history (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  connection_id UUID NOT NULL,
  status VARCHAR(20) NOT NULL, -- 'success', 'failed'
  error_message TEXT,
  connected_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (connection_id) REFERENCES ssh_connections(id) ON DELETE CASCADE
);

-- Index for faster lookups
CREATE INDEX idx_ssh_history_connection_id ON ssh_connection_history(connection_id);
CREATE INDEX idx_ssh_history_status ON ssh_connection_history(status);
```

## Encryption Strategy

All sensitive SSH credentials will be encrypted before storage:

1. **Encryption Key Management**:
   - Generate a unique encryption key for each user
   - Store the encryption key securely (consider using a key management service)
   - Alternatively, derive encryption key from user's password using PBKDF2
2. **Encryption Algorithm**:
   - Use AES-256-GCM for encryption
   - Store initialization vector (IV) alongside encrypted data
   - Include authentication tag to verify data integrity
3. **Implementation**:

```
// Example encryption function
async function encryptCredential(plaintext, userKey) {
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv('aes-256-gcm', userKey, iv);

  let encrypted = cipher.update(plaintext, 'utf8', 'base64');
  encrypted += cipher.final('base64');

  const authTag = cipher.getAuthTag().toString('base64');

  // Store IV, encrypted data, and auth tag together
  return {
    iv: iv.toString('base64'),
    data: encrypted,
    tag: authTag
  };
}

// Example decryption function
async function decryptCredential(encryptedData, userKey) {
  const iv = Buffer.from(encryptedData.iv, 'base64');
  const decipher = crypto.createDecipheriv('aes-256-gcm', userKey, iv);
  decipher.setAuthTag(Buffer.from(encryptedData.tag, 'base64'));

  let decrypted = decipher.update(encryptedData.data, 'base64', 'utf8');
  decrypted += decipher.final('utf8');

  return decrypted;
}
```

# API Endpoints

## SSH Connection Management

1. **Create SSH Connection**
   - Endpoint: `POST /api/ssh/connections`
   - Request Body:

```
{
  "name": "Production Server",
  "host": "example.com",
  "port": 22,
  "username": "admin",
  "authType": "password",
  "password": "securepassword",
  "privateKey": null,
```

```
  "passphrase": null
}
```

- Response:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Production Server",
  "host": "example.com",
  "port": 22,
  "username": "admin",
  "authType": "password",
  "lastConnected": null,
  "isActive": true,
  "createdAt": "2024-06-28T12:00:00Z",
  "updatedAt": "2024-06-28T12:00:00Z"
}
```

2. **List SSH Connections**
   - Endpoint: `GET /api/ssh/connections`
   - Response:

```
{
  "connections": [
    {
      "id": "550e8400-e29b-41d4-a716-446655440000",
      "name": "Production Server",
      "host": "example.com",
      "port": 22,
      "username": "admin",
      "authType": "password",
      "lastConnected": "2024-06-28T12:30:00Z",
      "isActive": true,
      "createdAt": "2024-06-28T12:00:00Z",
      "updatedAt": "2024-06-28T12:00:00Z"
    }
  ]
}
```

3. **Get SSH Connection Details**
   - Endpoint: `GET /api/ssh/connections/:id`
   - Response:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Production Server",
  "host": "example.com",
  "port": 22,
  "username": "admin",
  "authType": "password",
  "fingerprint": "SHA256:abcdefghijklmnopqrstuvwxyz1234567890ABCDEF",
  "lastConnected": "2024-06-28T12:30:00Z",
  "isActive": true,
  "createdAt": "2024-06-28T12:00:00Z",
  "updatedAt": "2024-06-28T12:00:00Z",
  "connectionHistory": [
    {
      "status": "success",
      "connectedAt": "2024-06-28T12:30:00Z"
    },
```

```
    {
      "status": "failed",
      "errorMessage": "Authentication failed",
      "connectedAt": "2024-06-28T12:15:00Z"
    }
  ]
}
```

4. **Update SSH Connection**
   - Endpoint: `PUT /api/ssh/connections/:id`
   - Request Body:

```
{
  "name": "Production Server Updated",
  "host": "example.com",
  "port": 2222,
  "username": "admin",
  "authType": "key",
  "privateKey": "-----BEGIN RSA PRIVATE KEY-----\n...\n-----END RSA PRIVATE KEY-
        ----",
  "passphrase": "keypassphrase"
}
```

   - Response:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Production Server Updated",
  "host": "example.com",
  "port": 2222,
  "username": "admin",
  "authType": "key",
  "lastConnected": "2024-06-28T12:30:00Z",
  "isActive": true,
  "createdAt": "2024-06-28T12:00:00Z",
  "updatedAt": "2024-06-28T13:00:00Z"
}
```

5. **Delete SSH Connection**
   - Endpoint: `DELETE /api/ssh/connections/:id`
   - Response:

```
{
  "message": "SSH connection deleted successfully"
}
```

## SSH Connection Testing

1. **Test SSH Connection**
   - Endpoint: `POST /api/ssh/connections/:id/test`
   - Response:

```
{
  "success": true,
  "message": "Connection successful",
  "serverInfo": {
    "platform": "Linux",
    "release": "5.10.0-15-amd64",
```

```
    "uptime": 1234567,
    "hostname": "server-hostname"
  }
}
```

### MCP Server Management

1. **Install MCP Server**
   - Endpoint: `POST /api/ssh/connections/:id/install-mcp`
   - Request Body:

```
{
  "version": "latest",
  "installPath": "/opt/mcp",
  "autoStart": true
}
```

   - Response:

```
{
  "success": true,
  "message": "MCP server installation started",
  "installationId": "550e8400-e29b-41d4-a716-446655440001"
}
```

2. **Check MCP Installation Status**
   - Endpoint: `GET /api/ssh/connections/:id/install-mcp/:installationId`
   - Response:

```
{
  "status": "completed",
  "progress": 100,
  "message": "MCP server installed successfully",
  "mcpUrl": "http://example.com:8080",
  "logs": ["Installation step 1 completed", "Installation step 2 completed"]
}
```

## Implementation Details

### Backend Implementation

1. **SSH Client Library**
   - Use `ssh2` npm package for SSH connections
   - Implement connection pooling for performance
   - Handle connection timeouts and retries
2. **Encryption Service**
   - Create a dedicated service for credential encryption/decryption
   - Implement key rotation capabilities
   - Add audit logging for security events
3. **MCP Installation Script**
   - Create a bash script for MCP installation
   - Support different Linux distributions
   - Include validation and error handling
   - Implement rollback capability for failed installations

### Frontend Implementation

1. **SSH Connection Form**
   - Create a form for adding/editing SSH connections
   - Implement field validation
   - Add support for uploading private key files
   - Include option to generate new key pairs
2. **Connection Testing UI**
   - Add a “Test Connection” button
   - Show real-time connection status
   - Display server information upon successful connection
   - Show detailed error messages for failed connections
3. **MCP Installation UI**
   - Create a wizard for MCP installation
   - Show installation progress in real-time
   - Display installation logs
   - Provide configuration options

## Security Considerations

1. **Credential Storage**
   - Never store plaintext credentials
   - Use strong encryption for all sensitive data
   - Implement proper key management
2. **Connection Security**
   - Validate SSH host fingerprints
   - Support for SSH key authentication
   - Implement connection timeouts
3. **Access Control**
   - Restrict SSH connection access to authorized users
   - Implement role-based permissions for SSH operations
   - Add audit logging for all SSH activities
4. **Frontend Security**
   - Never expose private keys or passwords in the DOM
   - Implement proper form validation
   - Use HTTPS for all API requests

## Testing Plan

1. **Unit Tests**
   - Test encryption/decryption functions
   - Test SSH connection handling
   - Test database operations
2. **Integration Tests**
   - Test API endpoints
   - Test SSH connection with mock server
   - Test MCP installation process
3. **Security Tests**
   - Test encryption strength
   - Test access control
   - Test for common vulnerabilities

## Deployment Considerations

1. **Database Migration**
   - Create migration scripts for new tables
   - Add indexes for performance
   - Update database backup procedures
2. **Environment Variables**
   - Add new environment variables for encryption keys

    - Document required configuration
3. **Monitoring**
    - Add logging for SSH connections
    - Monitor failed connection attempts
    - Set up alerts for suspicious activities

## Future Enhancements

1. **SSH Key Management**
    - Add UI for generating SSH key pairs
    - Implement key rotation
    - Support for SSH certificates
2. **Command Execution**
    - Add UI for executing commands on remote servers
    - Implement command history
    - Add support for scheduled commands
3. **File Transfer**
    - Add SFTP support for file transfers
    - Implement drag-and-drop file upload
    - Add support for directory synchronization
4. **Multi-Server Operations**
    - Support for executing commands on multiple servers
    - Implement server groups
    - Add parallel execution capabilities