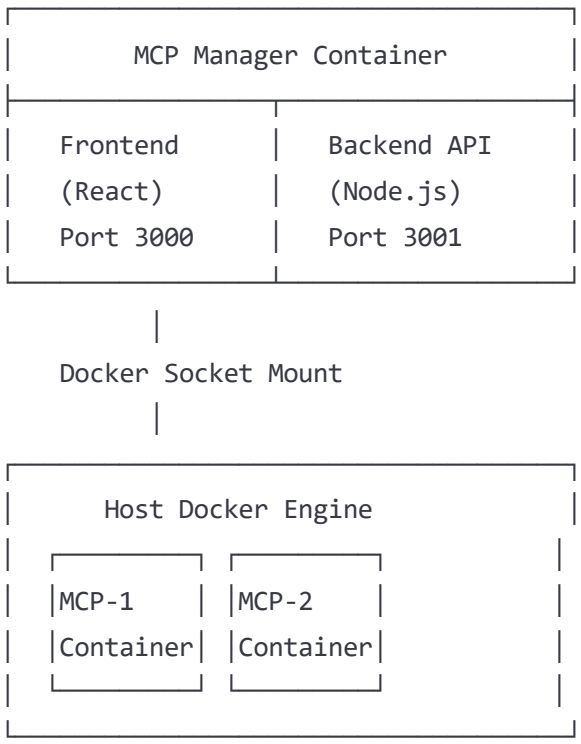
MCP Manager - Phase 1 PRD: Core Foundation

Timeline: Weeks 1-4

Goal: Establish basic Docker container with web UI and core server discovery/management

Success Criteria: Can discover, register, and start/stop containerized MCP servers

Phase 1 Architecture Overview



Technical Stack

Backend

- **Runtime**: Node.js 18+
- **Framework**: Express.js
- **Database**: SQLite (single file)
- **Docker Integration**: dockerode library
- **WebSocket**: ws library for real-time updates

Frontend

- **Framework**: React 18 with TypeScript
- **Styling**: Tailwind CSS

- **HTTP Client**: axios
- **WebSocket**: native WebSocket API

Infrastructure

- **Container**: Alpine Linux base
- **Process Manager**: Node.js native
- **Storage**: Volume mounts for persistence

Core Requirements

REQ-1.1: Project Setup and Infrastructure

Priority: P0
Estimated Effort: 8 hours

Functional Requirements:

- Docker container that runs MCP Manager
- Mounts Docker socket for container management
- Serves web UI on port 3000
- Exposes API on port 3001
- Persistent data storage via volume mounts

Technical Implementation:

dockerfile

```
FROM node:18-alpine

WORKDIR /app

# Install Docker CLI
RUN apk add --no-cache docker-cli

# Copy package files
COPY package*.json ./
RUN npm ci --production

# Copy application code
COPY . .

# Create non-root user
RUN addgroup -g 1001 mcpmanager && \
    adduser -S mcpmanager -u 1001 -G mcpmanager

# Create directories
RUN mkdir -p /app/data && \
    chown -R mcpmanager:mcpmanager /app

USER mcpmanager

EXPOSE 3000 3001

HEALTHCHECK --interval=30s --timeout=10s CMD curl -f http://localhost:3000/health || exit 1

CMD ["npm", "start"]
```

Database Schema:

```sql
-- SQLite schema
CREATE TABLE servers (
  id TEXT PRIMARY KEY DEFAULT (hex(randomblob(16))),
  name TEXT NOT NULL UNIQUE,
  type TEXT NOT NULL DEFAULT 'container', -- 'container' only in Phase 1
  image TEXT, -- Docker image name
  status TEXT NOT NULL DEFAULT 'stopped', -- 'running', 'stopped', 'error'
  port INTEGER, -- Exposed port
  container_id TEXT, -- Docker container ID
  config TEXT NOT NULL DEFAULT '{}', -- JSON configuration
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_servers_status ON servers(status);
CREATE INDEX idx_servers_type ON servers(type);
```

API Endpoints:

```javascript
// Health check
GET /health
Response: { status: 'ok', timestamp: ISO_DATE }

// Server management
GET /api/servers
POST /api/servers
GET /api/servers/:id
PUT /api/servers/:id
DELETE /api/servers/:id
POST /api/servers/:id/start
POST /api/servers/:id/stop
POST /api/servers/:id/restart

// System info
GET /api/system/info
```

Acceptance Criteria:

- [ ] Docker container builds successfully
- [ ] Container starts and serves web UI on port 3000

- [ ] API responds on port 3001
- [ ] Health check endpoint returns 200
- [ ] SQLite database initializes with correct schema
- [ ] Docker socket mount allows container listing

REQ-1.2: Basic Web UI Framework

Priority: P0
Estimated Effort: 12 hours

Functional Requirements:

- Responsive dashboard layout
- Navigation between pages
- Basic component library
- Error handling and loading states

Component Structure:

```
src/
├── components/
│   ├── Layout/
│   │   ├── Header.tsx
│   │   ├── Sidebar.tsx
│   │   └── Layout.tsx
│   ├── Common/
│   │   ├── Button.tsx
│   │   ├── Input.tsx
│   │   ├── Modal.tsx
│   │   ├── LoadingSpinner.tsx
│   │   └── StatusIndicator.tsx
│   └── Dashboard/
│       ├── ServerCard.tsx
│       ├── ServerList.tsx
│       └── Dashboard.tsx
├── pages/
│   ├── Dashboard.tsx
│   ├── Servers.tsx
│   └── Settings.tsx
├── hooks/
│   ├── useApi.ts
│   ├── useServers.ts
│   └── useWebSocket.ts
├── services/
│   ├── api.ts
│   └── websocket.ts
├── types/
│   ├── server.ts
│   └── api.ts
└── App.tsx
```

TypeScript Interfaces:

typescript

```typescript
// Core types
interface MCPServer {
  id: string;
  name: string;
  type: 'container';
  image: string;
  status: 'running' | 'stopped' | 'error';
  port?: number;
  containerId?: string;
  config: ServerConfig;
  createdAt: string;
  updatedAt: string;
}

interface ServerConfig {
  environment?: Record<string, string>;
  volumes?: string[];
  ports?: Record<string, string>;
}

interface ApiResponse<T> {
  success: boolean;
  data?: T;
  error?: string;
}
```

UI Components:

tsx

```tsx
// Status indicator component
const StatusIndicator: React.FC<{ status: MCPServer['status'] }> = ({ status }) => {
  const config = {
    running: { color: 'text-green-500', icon: '●', label: 'Running' },
    stopped: { color: 'text-gray-500', icon: '○', label: 'Stopped' },
    error: { color: 'text-red-500', icon: '⚠', label: 'Error' }
  };

  const { color, icon, label } = config[status];

  return (
    <span className={`flex items-center space-x-1 ${color}`}>
      <span>{icon}</span>
      <span className="text-sm font-medium">{label}</span>
    </span>
  );
};

// Server card component
const ServerCard: React.FC<{ server: MCPServer; onAction: (id: string, action: string) => void
  return (
    <div className="bg-white p-4 rounded-lg border border-gray-200 shadow-sm">
      <div className="flex items-center justify-between mb-2">
        <h3 className="text-lg font-medium text-gray-900">{server.name}</h3>
        <StatusIndicator status={server.status} />
      </div>
      <p className="text-sm text-gray-600 mb-3">{server.image}</p>
      <div className="flex space-x-2">
        {server.status === 'running' ? (
          <Button variant="danger" size="sm" onClick={() => onAction(server.id, 'stop')}>
            Stop
          </Button>
        ) : (
          <Button variant="success" size="sm" onClick={() => onAction(server.id, 'start')}>
            Start
          </Button>
        )}
        <Button variant="primary" size="sm" onClick={() => onAction(server.id, 'restart')}>
          Restart
        </Button>
      </div>
    </div>
```

```
    );
  };
```

Acceptance Criteria:

- [ ] Dashboard loads and displays correctly
- [ ] Navigation between pages works
- [ ] Components render with proper styling
- [ ] Loading states display during API calls
- [ ] Error messages show when API calls fail
- [ ] Responsive design works on mobile/tablet

REQ-1.3: Docker Integration Layer

Priority: P0

Estimated Effort: 16 hours

Functional Requirements:

- Connect to Docker daemon via socket
- List running containers
- Start/stop/restart containers
- Create new containers from images
- Remove containers

Technical Implementation:

```
javascript
```

```
// Docker service class
const Docker = require('dockerode');

class DockerService {
  constructor() {
    this.docker = new Docker({ socketPath: '/var/run/docker.sock' });
  }

  // List all MCP containers (filtered by label)
  async listMCPContainers() {
    try {
      const containers = await this.docker.listContainers({
        all: true,
        filters: {
          label: ['mcp.managed=true']
        }
      });

      return containers.map(container => ({
        id: container.Id,
        name: container.Names[0].replace('/', ''),
        image: container.Image,
        status: container.State,
        ports: container.Ports,
        created: container.Created
      }));
    } catch (error) {
      throw new Error(`Failed to list containers: ${error.message}`);
    }
  }

  // Create new MCP container
  async createContainer(config) {
    const containerConfig = {
      Image: config.image,
      name: config.name,
      Labels: {
        'mcp.managed': 'true',
        'mcp.server.id': config.serverId
      },
      ExposedPorts: config.port ? { [`${config.port}/tcp`]: {} } : {},
      Env: Object.entries(config.environment || {}).map(([key, value]) => `${key}=${value}`),
      HostConfig: {
```

```
      PortBindings: config.port ? {
        [`${config.port}/tcp`]: [{ HostPort: config.port.toString() }]
      } : {},
      RestartPolicy: { Name: 'unless-stopped' }
    }
  };

  try {
    const container = await this.docker.createContainer(containerConfig);
    return container.id;
  } catch (error) {
    throw new Error(`Failed to create container: ${error.message}`);
  }
}

// Container lifecycle operations
async startContainer(containerId) {
  try {
    const container = this.docker.getContainer(containerId);
    await container.start();
    return true;
  } catch (error) {
    throw new Error(`Failed to start container: ${error.message}`);
  }
}

async stopContainer(containerId) {
  try {
    const container = this.docker.getContainer(containerId);
    await container.stop({ t: 10 }); // 10 second timeout
    return true;
  } catch (error) {
    throw new Error(`Failed to stop container: ${error.message}`);
  }
}

async restartContainer(containerId) {
  try {
    const container = this.docker.getContainer(containerId);
    await container.restart({ t: 10 });
    return true;
  } catch (error) {
    throw new Error(`Failed to restart container: ${error.message}`);
  }
```

```
  }

  async removeContainer(containerId) {
    try {
      const container = this.docker.getContainer(containerId);
      await container.remove({ force: true });
      return true;
    } catch (error) {
      throw new Error(`Failed to remove container: ${error.message}`);
    }
  }

  // Get container info
  async getContainerInfo(containerId) {
    try {
      const container = this.docker.getContainer(containerId);
      const info = await container.inspect();

      return {
        id: info.Id,
        name: info.Name.replace('/', ''),
        status: info.State.Status,
        image: info.Config.Image,
        ports: info.NetworkSettings.Ports,
        created: info.Created,
        started: info.State.StartedAt
      };
    } catch (error) {
      throw new Error(`Failed to get container info: ${error.message}`);
    }
  }
}

module.exports = DockerService;
```

Server Management Service:

```
javascript
```

```
const DockerService = require('./DockerService');
const Database = require('./Database');

class ServerManager {
  constructor() {
    this.docker = new DockerService();
    this.db = new Database();
  }

  async createServer(serverData) {
    const serverId = generateId();

    try {
      // Create container
      const containerId = await this.docker.createContainer({
        ...serverData,
        serverId
      });

      // Save to database
      await this.db.createServer({
        id: serverId,
        name: serverData.name,
        type: 'container',
        image: serverData.image,
        status: 'stopped',
        port: serverData.port,
        container_id: containerId,
        config: JSON.stringify(serverData.config || {})
      });

      return { id: serverId, containerId };
    } catch (error) {
      throw new Error(`Failed to create server: ${error.message}`);
    }
  }

  async startServer(serverId) {
    try {
      const server = await this.db.getServer(serverId);
      if (!server) throw new Error('Server not found');

      await this.docker.startContainer(server.container_id);
```

```
    await this.db.updateServerStatus(serverId, 'running');

    return true;
  } catch (error) {
    await this.db.updateServerStatus(serverId, 'error');
    throw error;
  }
}

async stopServer(serverId) {
  try {
    const server = await this.db.getServer(serverId);
    if (!server) throw new Error('Server not found');

    await this.docker.stopContainer(server.container_id);
    await this.db.updateServerStatus(serverId, 'stopped');

    return true;
  } catch (error) {
    await this.db.updateServerStatus(serverId, 'error');
    throw error;
  }
}

async restartServer(serverId) {
  try {
    const server = await this.db.getServer(serverId);
    if (!server) throw new Error('Server not found');

    await this.docker.restartContainer(server.container_id);
    await this.db.updateServerStatus(serverId, 'running');

    return true;
  } catch (error) {
    await this.db.updateServerStatus(serverId, 'error');
    throw error;
  }
}

async syncServerStatus() {
  try {
    const servers = await this.db.getAllServers();

    for (const server of servers) {
```

```
        if (server.container_id) {
          try {
            const containerInfo = await this.docker.getContainerInfo(server.container_id);
            const newStatus = containerInfo.status === 'running' ? 'running' : 'stopped';

            if (server.status !== newStatus) {
              await this.db.updateServerStatus(server.id, newStatus);
            }
          } catch (error) {
            // Container doesn't exist
            await this.db.updateServerStatus(server.id, 'error');
          }
        }
      }
    } catch (error) {
      console.error('Failed to sync server status:', error);
    }
  }
}

module.exports = ServerManager;
```

Acceptance Criteria:

- [ ] Can connect to Docker daemon via socket
- [ ] Can list existing containers with MCP labels
- [ ] Can create new containers from image specifications
- [ ] Can start/stop/restart containers successfully
- [ ] Can retrieve container status and information
- [ ] Error handling works for Docker operation failures
- [ ] Database state syncs with actual container state

REQ-1.4: Server Discovery and Registration

Priority: P0
Estimated Effort: 12 hours

Functional Requirements:

- Auto-discover existing MCP containers
- Manual server registration form
- Server validation before creation

- Import/export server configurations

Discovery Service:

```
javascript
```

```
class ServerDiscoveryService {
  constructor(dockerService, database) {
    this.docker = dockerService;
    this.db = database;
  }

  async discoverExistingServers() {
    try {
      const containers = await this.docker.listMCPContainers();
      const discoveries = [];

      for (const container of containers) {
        // Check if already registered
        const existing = await this.db.getServerByContainerId(container.id);

        if (!existing) {
          const serverData = {
            name: container.name,
            image: container.image,
            containerId: container.id,
            status: container.status === 'running' ? 'running' : 'stopped',
            discovered: true
          };

          discoveries.push(serverData);
        }
      }

      return discoveries;
    } catch (error) {
      throw new Error(`Discovery failed: ${error.message}`);
    }
  }

  async registerDiscoveredServer(containerData) {
    const serverId = generateId();

    try {
      await this.db.createServer({
        id: serverId,
        name: containerData.name,
        type: 'container',
        image: containerData.image,
```

```
        status: containerData.status,
        container_id: containerData.containerId,
        config: JSON.stringify({})
      });

      return serverId;
    } catch (error) {
      throw new Error(`Failed to register server: ${error.message}`);
    }
  }

  validateServerConfig(config) {
    const errors = [];

    if (!config.name || config.name.trim().length === 0) {
      errors.push('Server name is required');
    }

    if (!config.image || config.image.trim().length === 0) {
      errors.push('Docker image is required');
    }

    if (config.port && (config.port < 1000 || config.port > 65535)) {
      errors.push('Port must be between 1000 and 65535');
    }

    if (config.name && !/^[a-zA-Z0-9-_]+$/.test(config.name)) {
      errors.push('Server name can only contain letters, numbers, hyphens, and underscores');
    }

    return {
      valid: errors.length === 0,
      errors
    };
  }
}
```

Server Registration UI:

tsx

```tsx
const ServerRegistrationForm: React.FC<{ onSubmit: (data: ServerConfig) => void; onCancel: () =
  const [formData, setFormData] = useState({
    name: '',
    image: '',
    port: '',
    environment: {} as Record<string, string>
  });
  const [errors, setErrors] = useState<string[]>([]);
  const [isSubmitting, setIsSubmitting] = useState(false);

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    setIsSubmitting(true);
    setErrors([]);

    try {
      const validation = validateServerConfig(formData);
      if (!validation.valid) {
        setErrors(validation.errors);
        return;
      }

      await onSubmit(formData);
    } catch (error) {
      setErrors([error.message]);
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <form onSubmit={handleSubmit} className="space-y-4">
      <div>
        <label className="block text-sm font-medium text-gray-700 mb-1">
          Server Name
        </label>
        <input
          type="text"
          value={formData.name}
          onChange={(e) => setFormData({ ...formData, name: e.target.value })}
          className="w-full px-3 py-2 border border-gray-300 rounded-md focus:outline-none focu
          placeholder="my-mcp-server"
          required
```

```
      />
    </div>

    <div>
      <label className="block text-sm font-medium text-gray-700 mb-1">
        Docker Image
      </label>
      <input
        type="text"
        value={formData.image}
        onChange={(e) => setFormData({ ...formData, image: e.target.value })}
        className="w-full px-3 py-2 border border-gray-300 rounded-md focus:outline-none focu
        placeholder="mcp/filesystem:latest"
        required
      />
    </div>

    <div>
      <label className="block text-sm font-medium text-gray-700 mb-1">
        Port (Optional)
      </label>
      <input
        type="number"
        value={formData.port}
        onChange={(e) => setFormData({ ...formData, port: e.target.value })}
        className="w-full px-3 py-2 border border-gray-300 rounded-md focus:outline-none focu
        placeholder="3000"
        min="1000"
        max="65535"
      />
    </div>

    {errors.length > 0 && (
      <div className="bg-red-50 border border-red-200 rounded-md p-3">
        <ul className="text-sm text-red-600 space-y-1">
          {errors.map((error, index) => (
            <li key={index}>• {error}</li>
          ))}
        </ul>
      </div>
    )}

    <div className="flex justify-end space-x-3">
      <Button variant="secondary" onClick={onCancel} disabled={isSubmitting}>
```

```
            Cancel
          </Button>
          <Button type="submit" variant="primary" disabled={isSubmitting}>
            {isSubmitting ? 'Creating...' : 'Create Server'}
          </Button>
        </div>
      </form>
    );
  };
```

Acceptance Criteria:

- [ ] Auto-discovery finds existing MCP containers
- [ ] Manual registration form validates input correctly
- [ ] Duplicate server names are prevented
- [ ] Invalid configurations show clear error messages
- [ ] Discovered servers can be imported with one click
- [ ] Server list updates immediately after registration

Testing Requirements

Unit Tests

javascript

```javascript
// Docker service tests
describe('DockerService', () => {
  test('should list MCP containers', async () => {
    const dockerService = new DockerService();
    const containers = await dockerService.listMCPContainers();
    expect(Array.isArray(containers)).toBe(true);
  });

  test('should create container with correct configuration', async () => {
    const dockerService = new DockerService();
    const config = {
      name: 'test-server',
      image: 'mcp/test:latest',
      port: 3000,
      environment: { TEST_VAR: 'value' }
    };

    const containerId = await dockerService.createContainer(config);
    expect(containerId).toBeDefined();
  });
});

// Server manager tests
describe('ServerManager', () => {
  test('should create and start server', async () => {
    const manager = new ServerManager();
    const serverData = {
      name: 'test-server',
      image: 'mcp/test:latest'
    };

    const result = await manager.createServer(serverData);
    expect(result.id).toBeDefined();

    const started = await manager.startServer(result.id);
    expect(started).toBe(true);
  });
});
```

Integration Tests

```javascript
describe('API Integration', () => {
  test('POST /api/servers should create new server', async () => {
    const response = await request(app)
      .post('/api/servers')
      .send({
        name: 'test-server',
        image: 'mcp/test:latest',
        port: 3000
      });

    expect(response.status).toBe(201);
    expect(response.body.success).toBe(true);
    expect(response.body.data.id).toBeDefined();
  });

  test('POST /api/servers/:id/start should start server', async () => {
    // Create server first
    const createResponse = await request(app)
      .post('/api/servers')
      .send({ name: 'test-server', image: 'mcp/test:latest' });

    const serverId = createResponse.body.data.id;

    // Start server
    const startResponse = await request(app)
      .post(`/api/servers/${serverId}/start`);

    expect(startResponse.status).toBe(200);
    expect(startResponse.body.success).toBe(true);
  });
});
```

Deployment Configuration

Docker Compose (Development)

yaml

```yaml
version: '3.8'
services:
  mcp-manager:
    build: .
    ports:
      - "3000:3000"
      - "3001:3001"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./data:/app/data
    environment:
      - NODE_ENV=development
      - LOG_LEVEL=debug
      - DATABASE_URL=sqlite:///app/data/mcp-manager.db
    restart: unless-stopped
```

Environment Variables

bash

```bash
# Application
NODE_ENV=development
PORT=3000
API_PORT=3001
LOG_LEVEL=debug

# Database
DATABASE_URL=sqlite:///app/data/mcp-manager.db

# Docker
DOCKER_SOCKET=/var/run/docker.sock

# Security
JWT_SECRET=your-jwt-secret-here
SESSION_SECRET=your-session-secret-here
```

Success Metrics

Functional Metrics

- [ ] Can discover existing MCP containers (100% success rate)
- [ ] Can create new containers from UI (100% success rate)

- [ ] Can start/stop containers reliably (>99% success rate)
- [ ] UI loads in <2 seconds
- [ ] API responses in <500ms average

Quality Metrics

- [ ] >90% test coverage
- [ ] Zero critical security vulnerabilities
- [ ] Docker container builds successfully
- [ ] All acceptance criteria met

Known Limitations

1. **Container-only support** - No traditional process management yet
2. **No authentication** - Security added in later phases
3. **Basic error handling** - Enhanced monitoring in Phase 2
4. **No health monitoring** - Status checking is container-level only
5. **Local deployment only** - Remote Docker support in later phases

Dependencies

Required for Development

- Docker Engine 20.0+
- Node.js 18+
- npm or yarn
- Git

External Dependencies

- dockerode (Docker API client)
- express (HTTP server)
- sqlite3 (Database)
- ws (WebSocket server)
- React 18 (Frontend framework)
- TypeScript (Type safety)

Next Phase Handoff

Upon completion of Phase 1, the following should be ready for Phase 2:

- Working Docker container with basic UI
- Container management functionality
- Database schema and API structure
- Basic component library
- Development environment setup
- Test framework established

Phase 2 will build upon this foundation to add traditional process management, health monitoring, and logging capabilities.