PSVirtualEnv PowerShell Module - Design Document

1. Proposed Module Name and Structure

Module Identity

- **Module Name**: `PSVirtualEnv`
- **Module File**: `PSVirtualEnv.psm1`
- **Manifest File**: `PSVirtualEnv.psd1`
- **Version**: 1.0.0 (initial release)

Module Directory Structure

```
PSVirtualEnv/
├── PSVirtualEnv.psd1          # Module manifest
├── PSVirtualEnv.psm1          # Main module file
├── Private/                   # Private functions
│   ├── PathManager.ps1        # PSModulePath manipulation
│   ├── ConfigManager.ps1      # Environment configuration
│   └── ValidationHelpers.ps1  # Input validation
├── Public/                    # Public cmdlets
│   ├── New-PSVirtualEnv.ps1
│   ├── Remove-PSVirtualEnv.ps1
│   ├── Activate-PSVirtualEnv.ps1
│   ├── Deactivate-PSVirtualEnv.ps1
│   ├── Get-PSVirtualEnv.ps1
│   ├── Install-PSModuleInEnv.ps1
│   ├── Uninstall-PSModuleInEnv.ps1
│   ├── Get-PSModuleInEnv.ps1
│   └── Update-PSModuleInEnv.ps1
├── Classes/                   # PowerShell classes
│   └── PSVirtualEnvironment.ps1
├── Data/                      # Static data
│   └── DefaultConfig.json
└── Tests/                     # Pester tests
    └── PSVirtualEnv.Tests.ps1
```

PSVirtualEnv Environment Structure

Each virtual environment will have this structure:

```
MyEnvironment/
├── config.json          # Environment metadata
├── Modules/             # Installed modules
│   ├── ModuleA/
│   └── ModuleB/
├── Scripts/             # Environment-specific scripts
├── Cache/               # Module cache and metadata
└── Logs/                # Operation logs
```

2. Detailed Cmdlet Design

New-PSVirtualEnv

Synopsis: Creates a new PowerShell virtual environment.

Parameters:

- `Name` (String, Mandatory): Name of the virtual environment
- `Path` (String, Optional): Custom path for the environment (defaults to `$env:USERPROFILE\.psvirtualenv\{Name}`)
- `CopySystemModules` (Switch, Optional): Include system modules in the environment
- `BaseEnvironment` (String, Optional): Copy modules from another PSVirtualEnv
- `Force` (Switch, Optional): Overwrite existing environment
- `Description` (String, Optional): Description of the environment

Example Usage:

```powershell
New-PSVirtualEnv -Name "WebProject" -Description "Environment for web development modules"
New-PSVirtualEnv -Name "Testing" -Path "C:\Projects\TestEnv" -CopySystemModules
```

Core Logic:

1. Validate environment name and path
2. Create directory structure
3. Initialize config.json with metadata
4. Optionally copy system modules or base environment modules
5. Register environment in global registry

Remove-PSVirtualEnv

Synopsis: Removes an existing PowerShell virtual environment.

Parameters:

- `Name` (String, Mandatory): Name of the environment to remove
- `Force` (Switch, Optional): Skip confirmation prompt

Example Usage:

```powershell
Remove-PSVirtualEnv -Name "OldProject"
Remove-PSVirtualEnv -Name "Testing" -Force
```

Core Logic:

1. Validate environment exists
2. Check if environment is currently active
3. Prompt for confirmation unless -Force is used
4. Remove environment directory and files
5. Unregister from global registry

Activate-PSVirtualEnv

Synopsis: Activates a PowerShell virtual environment in the current session.

Parameters:

- `Name` (String, Mandatory): Name of the environment to activate
- `Scope` (String, Optional): Scope of activation (Session, Global) - defaults to Session

Example Usage:

```powershell
Activate-PSVirtualEnv -Name "WebProject"
Activate-PSVirtualEnv -Name "Testing" -Scope Global
```

Core Logic:

1. Validate environment exists
2. Store current PSModulePath in session variable
3. Modify PSModulePath to prioritize environment modules
4. Update prompt function to show active environment
5. Set session variable tracking active environment

Deactivate-PSVirtualEnv

Synopsis: Deactivates the currently active PowerShell virtual environment.

Parameters:

- None (operates on currently active environment)

Example Usage:

```powershell
Deactivate-PSVirtualEnv
```

Core Logic:

1. Check if any environment is active
2. Restore original PSModulePath
3. Reset prompt function
4. Clear session tracking variables
5. Unload environment-specific modules if needed

Get-PSVirtualEnv

Synopsis: Lists PowerShell virtual environments.

Parameters:

- `Name` (String, Optional): Filter by specific environment name
- `Active` (Switch, Optional): Show only active environment
- `Detailed` (Switch, Optional): Show detailed information

Example Usage:

```powershell
Get-PSVirtualEnv
Get-PSVirtualEnv -Active
Get-PSVirtualEnv -Name "Web*" -Detailed
```

Core Logic:

1. Read global environment registry
2. Filter by name pattern if specified
3. Show active status and detailed info if requested

4. Return formatted environment objects

Install-PSModuleInEnv

Synopsis: Installs a PowerShell module into the active virtual environment.

Parameters:

- `Name` (String, Mandatory): Module name to install
- `RequiredVersion` (String, Optional): Specific version to install
- `Repository` (String, Optional): Repository to install from (defaults to PSGallery)
- `Scope` (String, Optional): Always set to CurrentUser for environment isolation
- `Force` (Switch, Optional): Force installation
- `AllowPrerelease` (Switch, Optional): Allow prerelease versions

Example Usage:

```powershell
Install-PSModuleInEnv -Name "Pester" -RequiredVersion "5.3.0"
Install-PSModuleInEnv -Name "PSScriptAnalyzer" -Force
```

Core Logic:

1. Verify environment is active
2. Set installation path to environment Modules directory
3. Call Install-Module with modified parameters
4. Update environment configuration
5. Log installation details

Uninstall-PSModuleInEnv

Synopsis: Uninstalls a module from the active virtual environment.

Parameters:

- `Name` (String, Mandatory): Module name to uninstall
- `RequiredVersion` (String, Optional): Specific version to uninstall
- `Force` (Switch, Optional): Force uninstallation

Example Usage:

```powershell
```

```
Uninstall-PSModuleInEnv -Name "OldModule"
Uninstall-PSModuleInEnv -Name "TestModule" -RequiredVersion "1.0.0" -Force
```

Core Logic:

1. Verify environment is active
2. Check if module exists in environment
3. Remove module from environment Modules directory
4. Update environment configuration
5. Log uninstallation details

Get-PSModuleInEnv

Synopsis: Lists modules installed in the active virtual environment.

Parameters:

- `Name` (String, Optional): Filter by module name pattern
- `ListAvailable` (Switch, Optional): Show all available modules in environment

Example Usage:

```powershell
Get-PSModuleInEnv
Get-PSModuleInEnv -Name "Pester*"
Get-PSModuleInEnv -ListAvailable
```

Core Logic:

1. Verify environment is active
2. Scan environment Modules directory
3. Filter by name pattern if specified
4. Return module information objects

Update-PSModuleInEnv

Synopsis: Updates modules in the active virtual environment.

Parameters:

- `Name` (String, Optional): Specific module to update (defaults to all)
- `Force` (Switch, Optional): Force update
- `AcceptLicense` (Switch, Optional): Accept license agreements

Example Usage:

```powershell
Update-PSModuleInEnv
Update-PSModuleInEnv -Name "Pester" -Force
```

Core Logic:

1. Verify environment is active
2. Check for module updates
3. Update specified or all modules
4. Update environment configuration
5. Log update details

3. Technical Approach Discussion

Chosen Strategy: PSModulePath Manipulation with Session Isolation

Primary Approach: Direct manipulation of `$env:PSModulePath` combined with session-scoped state management.

Rationale:

- **Simplicity**: Leverages existing PowerShell module loading mechanisms
- **Compatibility**: Works with all existing PowerShell cmdlets and functions
- **Performance**: Minimal overhead compared to custom runspaces
- **Transparency**: Users can still use standard PowerShell commands

Implementation Details:

1. **Path Management Strategy**:

   ```powershell
   # Store original path
   $script:OriginalPSModulePath = $env:PSModulePath

   # Modify path to prioritize environment
   $env:PSModulePath = "$EnvironmentPath\Modules;$SystemModulePath"
   ```

2. **Session State Tracking**:

   ```powershell
   ```

```
# Track active environment in session
$script:ActiveEnvironment = @{
    Name = $EnvironmentName
    Path = $EnvironmentPath
    OriginalPath = $OriginalPSModulePath
}
```

3. **Module Installation Redirection**:

```
powershell

# Override default installation path
$InstallPath = Join-Path $ActiveEnvironment.Path "Modules"
Install-Module -Name $ModuleName -Scope CurrentUser -Force -Path $InstallPath
```

Alternative Approaches Considered

1. Custom Runspaces:

- **Pros**: Complete isolation, no path manipulation
- **Cons**: Complex implementation, compatibility issues, performance overhead
- **Verdict**: Too complex for initial implementation

2. Module Proxying/Shimming:

- **Pros**: Fine-grained control over module loading
- **Cons**: Requires wrapping all module-related cmdlets, maintenance burden
- **Verdict**: Overly complex for the benefits provided

3. PowerShell Classes and Custom Loaders:

- **Pros**: Object-oriented approach, extensible
- **Cons**: Requires PowerShell 5.0+, complex state management
- **Verdict**: Good for future enhancement, not core implementation

PSModulePath Modification Strategy

Safe Modification Process:

1. **Backup**: Store original PSModulePath in module-scoped variable
2. **Validate**: Ensure environment path exists and is accessible
3. **Modify**: Prepend environment path to PSModulePath
4. **Track**: Maintain session state for active environment
5. **Restore**: Revert to original path on deactivation

Restoration Reliability:

- Use try-finally blocks for critical operations
- Implement session cleanup on module removal
- Provide manual restoration commands for edge cases

Integration with Standard Cmdlets

Install-Module Integration:

- Intercept and redirect installation path
- Maintain compatibility with all Install-Module parameters
- Update environment manifest after installation

Import-Module Integration:

- Leverage natural PowerShell module loading
- No modification needed due to PSModulePath manipulation
- Environment modules automatically take precedence

4. Configuration and Persistence

Environment Configuration (config.json)

```json
{
  "name": "WebProject",
  "path": "C:\\Users\\User\\.psvirtualenv\\WebProject",
  "created": "2024-01-15T10:30:00Z",
  "description": "Environment for web development modules",
  "modules": [
    {
      "name": "Pester",
      "version": "5.3.0",
      "installed": "2024-01-15T10:35:00Z"
    }
  ],
  "settings": {
    "includeSystemModules": false,
    "autoActivate": false
  }
}
```

Global Registry

- Location: `$env:USERPROFILE\.psvirtualenv\registry.json`
- Purpose: Track all environments and their metadata
- Format: JSON array of environment configurations

5. User Experience Enhancements

Tab Completion

- Environment names for activation/deactivation
- Module names within active environment
- Path completion for environment creation

Prompt Modification

```powershell
# Modified prompt to show active environment
function prompt {
    $env = Get-ActivePSVirtualEnv
    if ($env) {
        Write-Host "($($env.Name)) " -NoNewline -ForegroundColor Green
    }
    "PS $($executionContext.SessionState.Path.CurrentLocation)> "
}
```

Verbose Output

- Clear status messages during operations
- Progress indicators for long-running tasks
- Detailed error messages with suggested solutions

6. Challenges and Limitations

Implementation Challenges

1. Cross-Platform Compatibility:

- Path separators differ between Windows and Unix-like systems
- Module installation paths vary by platform
- **Solution**: Use PowerShell's cross-platform path handling

2. PowerShell Version Differences:

- Windows PowerShell vs PowerShell Core differences
- Module compatibility across versions

- **Solution**: Target PowerShell 5.1+ with compatibility shims

3. Module Dependencies:

- Complex dependency chains may span environments
- Circular dependencies in isolated environments
- **Solution**: Implement dependency resolution with user guidance

4. Performance Implications:

- Path manipulation affects module discovery performance
- Large numbers of environments could slow operations
- **Solution**: Implement caching and lazy loading

Inherent Limitations

1. Process Isolation:

- Cannot provide true process-level isolation
- Global variables and session state still shared
- **Limitation**: Document as known constraint

2. Native Binaries:

- Cannot isolate native DLLs and executables
- PATH environment variable not managed
- **Limitation**: Focus on PowerShell modules only

3. System Module Conflicts:

- System modules may still conflict with environment modules
- Administrative privileges may be required for some operations
- **Limitation**: Document best practices for handling

7. Error Handling Strategy

Comprehensive Error Handling

- Validate all inputs with clear error messages
- Implement rollback mechanisms for failed operations
- Provide troubleshooting guidance in error messages

Example Error Scenarios

```
powershell
```

```
# Environment not found
Write-Error "Environment 'NonExistent' not found. Use Get-PSVirtualEnv to list available environments."

# Module installation failure
Write-Error "Failed to install module 'BadModule'. Check repository availability and module name."

# Permission issues
Write-Error "Access denied creating environment at path. Try running as administrator or choose different path."
```

8. Testing Strategy

Unit Tests

- Test each cmdlet independently
- Mock external dependencies (Install-Module, file system)
- Validate parameter combinations and edge cases

Integration Tests

- Test complete workflows (create → activate → install → deactivate)
- Test cross-platform compatibility
- Performance testing with large numbers of environments

User Acceptance Tests

- Real-world scenarios with actual module installations
- Stress testing with complex dependency scenarios
- Documentation and help system validation

9. Future Enhancements

Advanced Features

- **Export/Import**: Save and share environment configurations
- **Dependency Resolution**: Automatic handling of module dependencies
- **Version Pinning**: Lock modules to specific versions
- **Environment Templates**: Predefined environment configurations
- **Integration**: VS Code extension for environment management

Performance Optimizations

- **Parallel Operations**: Multi-threaded module installations
- **Caching**: Module metadata caching for faster operations

- **Lazy Loading**: On-demand environment discovery

This design provides a solid foundation for implementing PSVirtualEnv while maintaining PowerShell idioms and ensuring robust functionality. The modular design allows for iterative development and future enhancements.