Plugin Development Guide

Overview

This guide explains how to develop custom plugins for the UBOM Workflow Engine's pluggable work execution system. Plugins allow you to extend the workflow engine with custom work execution logic while maintaining compatibility with the existing architecture.

Plugin Architecture

Plugin Interface

All plugins must implement the `ExternalWorkPlugin` interface:

```
type ExternalWorkPlugin interface {
    executors.EnhancedWorkExecutor

    // Plugin lifecycle methods
    Initialize(config map[string]interface{}) error
    Shutdown() error
    HealthCheck() error

    // Plugin information
    GetPluginInfo() PluginInfo
}
```

Base Plugin

The `BasePlugin` struct provides common functionality:

```
type BasePlugin struct {
    info     PluginInfo
    metadata executors.WorkMetadata
    schema   executors.WorkSchema
}
```

Step-by-Step Plugin Development

1. Define Plugin Information

Create a `PluginInfo` struct with metadata about your plugin:

```
info := plugins.PluginInfo{
    Name:        "my-custom-plugin",
    Version:     "1.0.0",
    Author:      "Your Name",
    Description: "Custom plugin for special work execution",
    WorkTypes:   []string{"custom"},
    APIVersion:  "1.0",
    Homepage:    "https://github.com/yourname/my-plugin",
    License:     "MIT",
}
```

2. Define Work Metadata

Specify metadata for the work types your plugin handles:

```
metadata := executors.WorkMetadata{
    Name:        "Custom Work Executor",
    Version:     "1.0.0",
    Author:      "Your Name",
    Description: "Executes custom work types",
    WorkTypes:   []string{"custom"},
}
```

3. Create JSON Schema

Define a JSON schema for validating work configurations:

```
const CustomWorkSchema = `{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "title": "Custom Work Configuration",
  "required": ["endpoint"],
  "properties": {
    "endpoint": {
      "type": "string",
      "description": "API endpoint to call"
    },
    "timeout": {
      "type": "string",
      "pattern": "^[0-9]+[smh]$",
      "default": "30s"
    }
  }
}`

schema := executors.WorkSchema{
    JSONSchema:    CustomWorkSchema,
    Examples:      getExamples(),
    Documentation: "Schema for custom work execution",
}
```

4. Implement Plugin Structure

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "time"

    "github.com/ubom/workflow/layer0"
    "github.com/ubom/workflow/executors"
    "github.com/ubom/workflow/plugins"
)

const WorkTypeCustom layer0.WorkType = "custom"

type CustomPlugin struct {
    *plugins.BasePlugin
    client HTTPClient
    logger Logger
}

type CustomConfig struct {
    Endpoint string        `json:"endpoint"`
    Timeout  time.Duration `json:"timeout"`
    Headers  map[string]string `json:"headers,omitempty"`
}
```

5. Implement Core Methods

Execute Method

```go
func (p *CustomPlugin) Execute(ctx context.Context, work layer0.Work, workContext *lay-
er0.Context) (executors.WorkResult, error) {
    startTime := time.Now()

    // Parse configuration
    config, err := p.parseConfig(work)
    if err != nil {
        return executors.WorkResult{
            Success: false,
            Error:   fmt.Sprintf("failed to parse config: %v", err),
            Metrics: executors.ExecutionMetrics{
                StartTime: startTime,
                EndTime:   time.Now(),
                Duration:  time.Since(startTime),
            },
        }, err
    }

    // Set timeout
    if config.Timeout > 0 {
        var cancel context.CancelFunc
        ctx, cancel = context.WithTimeout(ctx, config.Timeout)
        defer cancel()
    }

    // Execute custom logic
    response, logs, err := p.executeCustomLogic(ctx, config, work.GetInput())
    endTime := time.Now()

    result := executors.WorkResult{
        Logs: logs,
        Metrics: executors.ExecutionMetrics{
            StartTime: startTime,
            EndTime:   endTime,
            Duration:  endTime.Sub(startTime),
        },
    }

    if err != nil {
        result.Success = false
        result.Error = err.Error()
        return result, err
    }

    result.Success = true
    result.Outputs = map[string]interface{}{
        "response": response,
    }

    return result, nil
}
```

Validate Method

```
func (p *CustomPlugin) Validate(work layer0.Work) error {
    if work.GetType() != WorkTypeCustom {
        return fmt.Errorf("work type %s is not supported", work.GetType())
    }

    config, err := p.parseConfig(work)
    if err != nil {
        return fmt.Errorf("failed to parse config: %w", err)
    }

    return p.validateConfig(config)
}
```

Lifecycle Methods

```
func (p *CustomPlugin) Initialize(config map[string]interface{}) error {
    // Initialize HTTP client, connections, etc.
    p.client = NewHTTPClient()
    p.logger.Info("Custom plugin initialized")
    return nil
}

func (p *CustomPlugin) Shutdown() error {
    // Clean up resources
    if p.client != nil {
        p.client.Close()
    }
    p.logger.Info("Custom plugin shut down")
    return nil
}

func (p *CustomPlugin) HealthCheck() error {
    // Perform health check
    if p.client == nil {
        return fmt.Errorf("HTTP client not initialized")
    }
    return p.client.Ping()
}
```

Interface Compliance Methods

```
func (p *CustomPlugin) CanExecute(workType layer0.WorkType) bool {
    return workType == WorkTypeCustom
}

func (p *CustomPlugin) GetSupportedTypes() []layer0.WorkType {
    return []layer0.WorkType{WorkTypeCustom}
}
```

6. Plugin Factory Function

For dynamic loading, export a factory function:

```
// NewPlugin creates a new instance of the custom plugin
func NewPlugin() plugins.ExternalWorkPlugin {
    info := plugins.PluginInfo{
        Name:        "custom-plugin",
        Version:     "1.0.0",
        Author:      "Your Name",
        Description: "Custom work execution plugin",
        WorkTypes:   []string{string(WorkTypeCustom)},
        APIVersion:  "1.0",
    }

    metadata := executors.WorkMetadata{
        Name:        "Custom Plugin",
        Version:     "1.0.0",
        Author:      "Your Name",
        Description: "Executes custom work types",
        WorkTypes:   []string{string(WorkTypeCustom)},
    }

    schema := executors.WorkSchema{
        JSONSchema:    CustomWorkSchema,
        Examples:      getExamples(),
        Documentation: "Custom plugin for special work execution",
    }

    return &CustomPlugin{
        BasePlugin: plugins.NewBasePlugin(info, metadata, schema),
        logger:     NewLogger(),
    }
}
```

7. Helper Methods

```
func (p *CustomPlugin) parseConfig(work layer0.Work) (CustomConfig, error) {
    config := CustomConfig{
        Timeout: 30 * time.Second,
        Headers: make(map[string]string),
    }

    executorConfig, exists := work.GetConfiguration().Parameters["executor_config"]
    if !exists {
        return config, fmt.Errorf("executor_config not found")
    }

    configBytes, err := json.Marshal(executorConfig)
    if err != nil {
        return config, fmt.Errorf("failed to marshal config: %w", err)
    }

    if err := json.Unmarshal(configBytes, &config); err != nil {
        return config, fmt.Errorf("failed to unmarshal config: %w", err)
    }

    return config, nil
}

func (p *CustomPlugin) validateConfig(config CustomConfig) error {
    if config.Endpoint == "" {
        return fmt.Errorf("endpoint is required")
    }

    if config.Timeout <= 0 {
        return fmt.Errorf("timeout must be positive")
    }

    return nil
}

func (p *CustomPlugin) executeCustomLogic(ctx context.Context, config CustomConfig, in-
put interface{}) (interface{}, []executors.LogEntry, error) {
    logs := []executors.LogEntry{
        {
            Timestamp: time.Now(),
            Level:     "INFO",
            Message:   "Starting custom execution",
            Source:    "custom-plugin",
        },
    }

    // Implement your custom logic here
    response, err := p.client.Post(config.Endpoint, input, config.Headers)
    if err != nil {
        logs = append(logs, executors.LogEntry{
            Timestamp: time.Now(),
            Level:     "ERROR",
            Message:   fmt.Sprintf("Request failed: %v", err),
            Source:    "custom-plugin",
        })
        return nil, logs, err
    }

    logs = append(logs, executors.LogEntry{
        Timestamp: time.Now(),
        Level:     "INFO",
        Message:   "Custom execution completed",
```

```
        Source:    "custom-plugin",
    })

    return response, logs, nil
}

func getExamples() []executors.WorkDefinition {
    return []executors.WorkDefinition{
        {
            Name:        "Simple API Call",
            Description: "Make a simple API call",
            Configuration: map[string]interface{}{
                "executor_config": map[string]interface{}{
                    "endpoint": "https://api.example.com/process",
                    "timeout":  "30s",
                    "headers": map[string]string{
                        "Content-Type": "application/json",
                    },
                },
            },
            Input:          map[string]interface{}{"data": "test"},
            ExpectedOutput: map[string]interface{}{"response": "processed"},
        },
    }
}
```

Building and Deployment

1. Build as Shared Library

For dynamic loading, build your plugin as a shared library:

```
go build -buildmode=plugin -o custom-plugin.so main.go
```

2. Static Registration

For static registration, import and register your plugin:

```
import "path/to/your/plugin"

// Register plugin
loader := plugins.NewDefaultPluginLoader(logger)
factory := func() plugins.ExternalWorkPlugin {
    return plugin.NewPlugin()
}
err := loader.RegisterPlugin("custom-plugin", factory)
```

3. Configuration

Create a plugin configuration file:

```
{
  "plugins": [
    {
      "name": "custom-plugin",
      "enabled": true,
      "config": {
        "default_timeout": "30s",
        "max_retries": 3
      },
      "priority": 10
    }
  ]
}
```

Testing Your Plugin

Unit Tests

```
func TestCustomPlugin_Execute(t *testing.T) {
    plugin := NewPlugin().(*CustomPlugin)
    plugin.client = &MockHTTPClient{}

    work := createTestWork()
    ctx := context.Background()
    workContext := &layer0.Context{}

    result, err := plugin.Execute(ctx, work, workContext)
    if err != nil {
        t.Fatalf("Execute failed: %v", err)
    }

    if !result.Success {
        t.Errorf("Expected success, got failure: %s", result.Error)
    }
}
```

Integration Tests

```go
func TestCustomPlugin_Integration(t *testing.T) {
    loader := plugins.NewDefaultPluginLoader(&MockLogger{})

    factory := func() plugins.ExternalWorkPlugin {
        return NewPlugin()
    }

    err := loader.RegisterPlugin("custom-plugin", factory)
    if err != nil {
        t.Fatalf("Failed to register plugin: %v", err)
    }

    // Test plugin lifecycle
    err = loader.InitializePlugin("custom-plugin", map[string]interface{}{})
    if err != nil {
        t.Fatalf("Failed to initialize plugin: %v", err)
    }

    // Test execution
    plugin, err := loader.GetPlugin("custom-plugin")
    if err != nil {
        t.Fatalf("Failed to get plugin: %v", err)
    }

    // ... test execution logic
}
```

Best Practices

1. **Error Handling**: Provide detailed error messages and proper error types
2. **Resource Management**: Always clean up resources in the Shutdown method
3. **Configuration Validation**: Validate all configuration parameters
4. **Logging**: Use structured logging for better observability
5. **Testing**: Write comprehensive unit and integration tests
6. **Documentation**: Provide clear documentation and examples
7. **Versioning**: Use semantic versioning for your plugins
8. **Security**: Validate all inputs and sanitize outputs
9. **Performance**: Optimize for concurrent execution
10. **Compatibility**: Maintain backward compatibility when updating plugins

Common Patterns

HTTP Client Plugin

For plugins that make HTTP requests, use connection pooling and proper timeout handling.

Database Plugin

For database operations, use connection pooling and transaction management.

File Processing Plugin

For file operations, handle large files efficiently and provide progress reporting.

Message Queue Plugin

For message queue integration, handle connection failures and implement proper retry logic.

Troubleshooting

Common Issues

1. **Plugin Not Loading**: Check the plugin factory function signature
2. **Configuration Errors**: Validate JSON schema and configuration parsing
3. **Memory Leaks**: Ensure proper resource cleanup in Shutdown method
4. **Concurrency Issues**: Use proper synchronization for shared resources
5. **Performance Problems**: Profile your plugin and optimize bottlenecks

Debugging Tips

1. Use structured logging to trace execution flow
2. Implement comprehensive health checks
3. Add metrics collection for monitoring
4. Use proper error wrapping for better error context
5. Test with various input scenarios and edge cases