GRL API Client - Complete User Guide

Table of Contents

Introduction

The GRL API Client is a comprehensive tool for automated interaction with GRL testing applications. This guide covers everything you need to know about setup, configuration, and usage of the client.

Project Structure and Responsibilities

Root Directory

- **grl_config.json**: The main configuration file that controls the client's behavior. Contains paths, IP addresses, port numbers, and other global settings.
- **sample_run.py**: Example script that demonstrates the basic workflow of the client. Use this as a template for your own tests.

client/ Directory

This directory contains the high-level client interface that you'll interact with directly.

- **grl_api_client.py**: The main client class that provides a simplified API for interacting with GRL applications. It handles:
  - Application lifecycle (launch, shutdown)
  - Test equipment connection
  - Project setup and configuration

    - Test execution and monitoring
    - Popup message handling
- **system_state.py**: Contains the SystemState dataclass which tracks the current state of the application and test execution. Used internally by the client.

API/ Directory

This directory contains the low-level API implementation that communicates directly with the GRL application.

- **init.py**: Exports the GRLApiHandler class for easy importing.
- **grl_api_handler.py**: The main API handler that sends HTTP requests to the GRL application's API endpoints. Handles all direct communication with the application.
- **diagnostics_api_handler.py**: Specialized handler for diagnostics and health checks. Provides functionality for checking API connectivity and gathering system information.
- **decorators.py**: Contains utility decorators for API methods, such as logging and error handling.

JSON_User_input/ Directory

This directory contains JSON configuration models used to set up projects in the GRL application.

- **project_config.json**: Contains the ProjectConfigurationModel with basic project information like name and owner.
- **esdf.json**: Contains the EsdfConfigurationModel which defines test equipment-specific data format settings.
- **tester_config.json**: Contains the TesterConfigurationModel which defines tester hardware settings.
- **report_config.json**: Contains the ReportConfigurationModel which defines how test reports should be formatted.

utils/ Directory

This directory contains utility modules that provide supporting functionality.

- **config_manager.py**: Handles loading and parsing the grl_config.json file.
- **log_manager.py**: Provides logging functionality with file rotation and formatting options.
- **web_app_controller.py**: Manages launching and connecting to the GRL application process.

Generated Files and Directories

During execution, the following files and directories will be created:

- **project_Configuration_Model_input.json**: A combined configuration file created from the individual JSON_User_input files.

- **popup_messages.json**: A record of all popup messages encountered during execution.
- **test_case_popup_messages.json**: Popup messages organized by test case.
- **grl_api_debug_Selected_run.log** (or your configured log filename): Log file with detailed execution information.
- **Test_Case_List_From_System/**: Directory containing test case listings:
  - **Generated_Test_cases_list.json**: Default test case listing file.
  - **Test_cases_list_[ProjectName].json**: Project-specific test case listing (if enabled).

Configuration Guide

grl_config.json

This is the main configuration file for the GRL API Client. Here's a detailed explanation of each section:

```json
{
    "common": {
        "initial_wait": 10,                 // Seconds to wait after launching the app
        "default_log_mode": "a",            // Log mode: "a" for append, "w" for overwrite
        "max_connection_attempts": 3,       // Number of connection attempts
        "connection_timeout": 30,           // Connection timeout in seconds
        "api_timeout": 60,                  // API call timeout in seconds
        "log_filename": "grl_api_debug.log"  // Path to log file
    },
    "applications": {
        "GRL-C3-MP-TPR": {                  // Application identifier
            "app_path": "C:\\Program Files\\GRL\\GRL-C3-MP-TPR\\AppFiles\\C3BrowserApp_MPP_TPR.
            "known_port": 2002,             // Port the application uses
            "app_name": "GRL-C3-MP-TPR"     // Display name of the application
        }
    },
    "default_app": "GRL-C3-MP-TPR",         // Default application to use
    "load_from_json": "True",               // Whether to load config from JSON files
    "ip_address": "192.168.5.9",            // Default IP address for test equipment
    "project_name_with_time_stamp": "True"   // Add timestamp to project names
}
```

Project Configuration Files

The JSON_User_input directory contains four JSON files that define your project configuration:

1. **project_config.json**: Basic project information

```json
{
  "ProjectConfigurationModel": {
    "projectName": "MyProject",
    "testerId": "TestEngineer",
    "dut": "Device001"
  }
}
```

2. **esdf.json**: ESDF configuration

```json
{
  "EsdfConfigurationModel": {
    // ESDF-specific configuration values
  }
}
```

3. **tester_config.json**: Tester configuration

```json
{
  "TesterConfigurationModel": {
    // Tester-specific configuration values
  }
}
```

4. **report_config.json**: Report configuration

```json
{
  "ReportConfigurationModel": {
    // Report-specific configuration values
  }
}
```

Setup Instructions

1. **Install Dependencies**:

```
pip install requests
```

2. **Create Directory Structure**: Set up the directories as outlined in the Project Structure section above.
3. **Configure Files**:
   - Modify `grl_config.json` with your specific settings

- Create/update the JSON files in JSON_User_input to match your project needs

4. **Verify Setup**:
   - Run `sample_run.py` to verify the basic functionality works

Running Tests

Basic Workflow

The typical workflow for using the GRL API Client follows these steps:

1. **Initialize the Client**:

```python
from client.grl_api_client import GRLApiClient
client = GRLApiClient("grl_config.json")
```

2. **Launch the Application**:

```python
if client.launch_app():
    # Application launched successfully
    # Continue with next steps
else:
    # Handle launch failure
```

3. **Connect to Test Equipment**:

```python
# Use IP from config file
connection_result = client.connect()
# Or specify IP directly
connection_result = client.connect("192.168.5.53")

if "error" in connection_result:
    # Handle connection error
else:
    # Connected successfully
```

4. **Set Up Project**:

```python
# Use project name from JSON_User_input/project_config.json
client.set_project()
# Or override with a custom name
client.set_project("CustomProject")
```

5. **Define and Run Tests**:

```python
test_list = [
    "7.1 MPP.PTX.POW.Digital_Ping_128kHz_P1",
    "7.2 MPP.PTX.POW.Digital_Ping_360kHz_P1"
]
client.submit_test_list(test_list)
```

6. **Clean Up**:

```python
# Always disconnect at the end
client.disconnect()
```

Full Example

Here's a complete example that properly handles errors and cleanup:

python

```python
from client.grl_api_client import GRLApiClient

def run_test_suite():
    client = GRLApiClient("grl_config.json")

    try:
        # Launch application
        if not client.launch_app():
            print("Failed to launch application")
            return False

        # Connect to test equipment
        connection_result = client.connect()
        if "error" in connection_result:
            print(f"Connection failed: {connection_result['error']}")
            return False

        # Set up project
        client.set_project("TestProject")

        # Define tests
        tests = [
            "7.1 MPP.PTX.POW.Digital_Ping_128kHz_P1",
            "7.2 MPP.PTX.POW.Digital_Ping_360kHz_P1",
            "7.3 MPP.PTX_POW_Cloak_Ping_360_LPM_TC1"
        ]

        # Run tests
        result = client.submit_test_list(tests)
        if not result.get("success"):
            print(f"Test execution failed: {result.get('error', 'Unknown error')}")
            return False

        return True

    except Exception as e:
        print(f"Exception occurred: {str(e)}")
        return False

    finally:
        # Always disconnect
        client.disconnect()

if __name__ == "__main__":
    if run_test_suite():
        print("Test suite completed successfully")
```

```
    else:
        print("Test suite failed")
```

Advanced Features

Popup Handling

The client automatically handles popups that appear during test execution. All popup messages are saved to:

- `popup_messages.json` - A chronological list of all popup messages
- `test_case_popup_messages.json` - Popup messages organized by test case

You can examine these files after a test run to see what messages were displayed.

Monitoring Test Status

The client maintains a SystemState object that tracks the current state of the application and test execution. During test execution, it continuously polls the API to update this state.

You can access the current state through `client.system_state_data`.

Stopping Tests

To stop a test in progress:

```python
client.stop_test_execution()
```

Automated Test Suites

For running multiple test configurations, you can load test cases from files:

```python
import json

# Load test configuration
with open('test_config.json', 'r') as f:
    test_configurations = json.load(f)

# Run each configuration
for config_name, test_list in test_configurations.items():
    print(f"Running test configuration: {config_name}")
    client.submit_test_list(test_list)
```

Troubleshooting

Common Issues

1. **Application Launch Fails**:
   - Check the app_path in grl_config.json
   - Verify the application is installed and functional
   - Look for error messages in the log file
2. **Connection to Test Equipment Fails**:
   - Verify the IP address is correct
   - Check physical connections to the test equipment
   - Verify the test equipment is powered on and ready
3. **Test Execution Failures**:
   - Check for errors in the test case names
   - Look at popup_messages.json for any unexpected dialogs
   - Check the log file for detailed error information

Log Files

The log file (default: grl_api_debug_Selected_run.log) contains detailed information about the client's operation. Common log levels:

- **INFO**: Normal operation messages
- **DEBUG**: Detailed operation information
- **WARNING**: Potential issues that didn't cause failure
- **ERROR**: Problems that prevented successful operation

You can adjust the log level in the code:

```python
import logging
from client.grl_api_client import GRLApiClient

client = GRLApiClient("grl_config.json")
client.log_manager.set_log_level(logging.DEBUG)  # For more verbose logs
# or
client.log_manager.set_log_level(logging.INFO)   # For normal verbosity
```

Customization and Extension

Adding New API Methods

To add a new API method:

1. Add the method to the appropriate API handler in the API/ directory
2. Add a corresponding high-level method in GRLApiClient

Customizing Popup Handling

If you need custom popup handling, you can modify the `_handle_connection_popup` method in `grl_api_client.py`.

Custom Logging

To customize logging:

```python
client = GRLApiClient("grl_config.json")
client.log_manager.set_log_formatter('%(asctime)s - %(levelname)s - %(message)s')  # Simple for
# or
client.log_manager.use_predefined_format('detailed')  # Detailed format
```

Best Practices

1. **Always Use Try-Finally**: Always wrap your client usage in try-finally blocks to ensure proper disconnection.
2. **Check Return Values**: All methods return status information - check them to handle errors gracefully.
3. **Log Levels**: Use appropriate log levels during development vs. production.
4. **Test Case Names**: Ensure test case names match exactly what's in the system.
5. **Configuration Management**: Keep your JSON_User_input files under version control.
6. **Regular Updates**: Check for updates to the API definitions if the GRL application is updated.

Complete File Structure

Here's the complete file structure with descriptions of each file:

```
/
├── grl_config.json              # Main configuration file
├── sample_run.py                # Sample script demonstrating API usage
├── client/                      # High-level client interface
│   ├── grl_api_client.py        # Main API client implementation
│   └── system_state.py          # System state tracking dataclass
├── API/                         # Low-level API implementation
│   ├── __init__.py              # Exports GRLApiHandler
│   ├── grl_api_handler.py       # Main API handler implementation
│   ├── diagnostics_api_handler.py # Diagnostics functionality
│   └── decorators.py            # API method decorators
├── JSON_User_input/             # Project configuration JSON files
│   ├── project_config.json      # Project configuration model
│   ├── esdf.json                # ESDF configuration model
│   ├── tester_config.json       # Tester configuration model
│   └── report_config.json       # Report configuration model
└── utils/                       # Utility modules
    ├── config_manager.py        # Configuration management
    ├── log_manager.py           # Logging utilities
    └── web_app_controller.py    # Application process control

# Generated during execution:
/
├── project_Configuration_Model_input.json  # Combined project configuration
├── popup_messages.json                     # Recorded popup messages
├── test_case_popup_messages.json           # Popups organized by test case
├── grl_api_debug_Selected_run.log          # Log file (name from config)
└── Test_Case_List_From_System/             # Test case listings directory
    ├── Test_cases_list_[ProjectName].json  # Project-specific test listing
    └── Generated_Test_cases_list.json      # Default test case listing
```

Details of Key Files

client/grl_api_client.py

The main client class that provides high-level methods for:

- Launching and disconnecting from the GRL application
- Connecting to test equipment
- Setting up projects
- Running tests
- Handling popups

python

```python
# Key methods:
# - __init__(self, config_file_path) - Initialize with config
# - launch_app(self) -> bool - Launch the application
# - connect(self, ip_address=None) -> Dict - Connect to test equipment
# - set_project(self, project_name=None) -> bool - Set up a project
# - submit_test_list(self, test_list) -> Dict - Run test cases
# - disconnect(self) -> None - Disconnect and clean up
# - stop_test_execution(self) -> bool - Stop a running test
```

client/system_state.py

A dataclass that tracks the current state of the system:

python

```python
@dataclass
class SystemState:
    app_state: str  # 'BUSY', 'IDLE', etc.
    connection_state: str  # 'CONNECTED', 'DISCONNECTED', etc.
    test_case_name: Optional[str] = None  # Current test case
    test_status: Optional[str] = None  # Test status ('Started', etc.)
```

API/grl_api_handler.py

The main class that sends HTTP requests to the GRL application API:

python

```python
# Key methods:
# - send_request(method, service, endpoint, params, data, headers) - Base request method
# - get_software_version() - Get application version
# - connect_to_test_equipment(ip_address) - Connect to tester
# - post_test_list_to_execute(test_list) - Submit tests
# - get_test_status() - Check test status
# - get_app_state() - Get application state
# - post_force_stop() - Stop test execution
# - put_project_folder(project_data) - Create/update project
```

API/diagnostics_api_handler.py

Specialized handler for diagnostics and health checks:

```python
# Key methods:
# - check_api_health(use_parallel=False) - Check API endpoint health
# - log_api_diagnostics() - Run comprehensive diagnostics
```

utils/config_manager.py

Handles loading and parsing configuration:

```python
# Key methods:
# - load_config() - Load from grl_config.json
# - get_app_config(app_name) - Get app-specific config
```

utils/log_manager.py

Provides logging functionality:

```python
# Key methods:
# - get_logger() - Get configured logger
# - set_log_level(level) - Set logging level
# - log_run_start(include_system_info) - Log run start
# - set_log_formatter(format_string) - Set custom format
# - use_predefined_format(format_type) - Use builtin format
```

utils/web_app_controller.py

Manages launching and connecting to the application process:

```python
# Key methods:
# - start_and_get_url(initial_wait) - Launch and connect
# - stop_process() - Stop the application
# - is_running() - Check if application is running
```

Conclusion

The GRL API Client provides a robust interface for automating GRL testing applications. By following this guide, you can efficiently set up, configure, and run automated tests with comprehensive error handling and logging. The modular design allows for customization and extension as needed for your specific testing requirements.