"IOC_Config Documentation"

"Michele Bigi"

IOC_Config - API Reference Manual

Version: 1.2.0
Date: December 2, 2025
Status: Production Ready

Table of Contents

Core Classes

OopParser

Main class for configuration file management.

Namespace

```
namespace ioc_config {
    class OopParser;
}
```

Constructor/Destructor

```
OopParser();
```

- **Description**: Default constructor. Initializes empty configuration.
- **Parameters**: None
- **Return**: Instance of OopParser

```
~OopParser();
```

- **Description**: Destructor. Automatically cleans up resources.
- **Parameters**: None
- **Return**: N/A

File Loading Methods

```cpp
bool loadFromOop(const std::string& filepath);
```

- **Description**: Load configuration from OOP (Object-Oriented Properties) format file
- **Parameters**:
  - `filepath` (const std::string&): Path to .oop file
- **Return**: `true` if successful, `false` otherwise
- **Error**: Call `getLastError()` for details
- **Format Example**:

  ```
  [section].
  key = value
  ```

```cpp
bool loadFromJson(const std::string& filepath);
```

- **Description**: Load configuration from JSON format file
- **Parameters**:
  - `filepath` (const std::string&): Path to .json file
- **Return**: `true` if successful, `false` otherwise
- **Format Example**:

  ```json
  {
    "section": {
      "key": "value"
    }
  }
  ```

```cpp
bool loadFromXml(const std::string& filepath);
```

- **Description**: Load configuration from XML format file
- **Parameters**:
  - `filepath` (const std::string&): Path to .xml file
- **Return**: `true` if successful, `false` otherwise
- **Format Example**:

  ```xml
  <config>
    <section>
      <key>value</key>
    </section>
  </config>
  ```

```cpp
bool loadFromCsv(const std::string& filepath, bool hasHeader =
        true);
```

- **Description**: Load configuration from CSV (Comma-Separated Values) format file
- **Parameters**:
  - `filepath` (const std::string&): Path to .csv file
  - `hasHeader` (bool): Whether first row contains column names (default: true)
- **Return**: `true` if successful, `false` otherwise
- **Format Example** (with headers):

  ```
  section,key,value
  object,id,17030
  object,name,Vesta
  ```

```cpp
bool loadFromYaml(const std::string& filepath);
```

- **Description**: Load configuration from YAML format file (requires YAML support)
- **Parameters**:
  - `filepath` (const std::string&): Path to .yaml/.yml file
- **Return**: `true` if successful, `false` otherwise
- **Availability**: Only when compiled with YAML support (#ifdef WITH_YAML)

```cpp
bool loadFromToml(const std::string& filepath);
```

- **Description**: Load configuration from TOML format file (requires TOML support)
- **Parameters**:
  - `filepath` (const std::string&): Path to .toml file
- **Return**: `true` if successful, `false` otherwise
- **Availability**: Only when compiled with TOML support (#ifdef WITH_TOML)

```cpp
bool loadFromString(const std::string& content, const
        std::string& format);
```

- **Description**: Load configuration from string content
- **Parameters**:
  - `content` (const std::string&): Configuration content
  - `format` (const std::string&): Format type (“oop”, “json”, “xml”, “csv”, “yaml”, “toml”)
- **Return**: `true` if successful, `false` otherwise
- **Example**:

  ```cpp
  std::string json_content = R"({"section": {"key":
          "value"}})";
  parser.loadFromString(json_content, "json");
  ```

File Saving Methods

```cpp
bool saveToOop(const std::string& filepath);
```

- **Description**: Save configuration to OOP format file
- **Parameters**:
  - `filepath` (const std::string&): Output file path
- **Return**: `true` if successful, `false` otherwise
- **Note**: Creates parent directories if needed

```cpp
bool saveToJson(const std::string& filepath);
```

- **Description**: Save configuration to JSON format file
- **Parameters**:
  - `filepath` (const std::string&): Output file path
- **Return**: `true` if successful, `false` otherwise

```cpp
bool saveToXml(const std::string& filepath);
```

- **Description**: Save configuration to XML format file
- **Parameters**:
  - `filepath` (const std::string&): Output file path
- **Return**: `true` if successful, `false` otherwise

```cpp
bool saveToYaml(const std::string& filepath);
```

- **Description**: Save configuration to YAML format file
- **Parameters**:
  - `filepath` (const std::string&): Output file path
- **Return**: `true` if successful, `false` otherwise
- **Availability**: Only with YAML support

```cpp
bool saveToToml(const std::string& filepath);
```

- **Description**: Save configuration to TOML format file
- **Parameters**:
  - `filepath` (const std::string&): Output file path
- **Return**: `true` if successful, `false` otherwise
- **Availability**: Only with TOML support

String Output Methods

```cpp
std::string saveToOopString() const;
```

- **Description**: Serialize configuration to OOP format string
- **Parameters**: None
- **Return**: Configuration in OOP format
- **Exception Safety**: No exceptions

```cpp
std::string saveToJsonString() const;
```

- **Description**: Serialize configuration to JSON format string
- **Parameters**: None
- **Return**: Configuration in JSON format

- **Exception Safety**: No exceptions

```
std::string saveToXmlString() const;
```

- **Description**: Serialize configuration to XML format string
- **Parameters**: None
- **Return**: Configuration in XML format

Data Access Methods

```
ConfigSectionData* getSection(const std::string& name);
```

- **Description**: Retrieve section by name
- **Parameters**:
  - `name` (const std::string&): Section name
- **Return**: Pointer to ConfigSectionData or nullptr if not found
- **Thread Safety**: Protected by mutex
- **Lifetime**: Pointer valid only while OopParser exists

```
ConfigParameter* getParameter(const std::string& section,
                              const std::string& key);
```

- **Description**: Retrieve parameter by section and key
- **Parameters**:
  - `section` (const std::string&): Section name
  - `key` (const std::string&): Parameter key
- **Return**: Pointer to ConfigParameter or nullptr if not found
- **Thread Safety**: Protected by mutex

```
std::string getParameter(const std::string& section,
                         const std::string& key,
                         const std::string& defaultValue) const;
```

- **Description**: Retrieve parameter with fallback to default value
- **Parameters**:
  - `section` (const std::string&): Section name
  - `key` (const std::string&): Parameter key
  - `defaultValue` (const std::string&): Returned if parameter not found
- **Return**: Parameter value or defaultValue
- **Thread Safety**: Protected by mutex

```
std::string getValueByPath(const std::string& path) const;
```

- **Description**: Retrieve value using RFC 6901 JSON Pointer path

- **Parameters**:

  - `path` (const std::string&): Path like “/section/parameter”

- **Return**: Value at path or empty string if not found

- **Example**:

  ```
  std::string id = parser.getValueByPath("/object/id");
  ```

- **Thread Safety**: Protected by mutex

Data Modification Methods

```
bool setParameter(const std::string& section,
                  const std::string& key,
                  const std::string& value);
```

- **Description**: Set parameter value (creates section if needed)
- **Parameters**:
  - `section` (const std::string&): Section name (created if not exists)
  - `key` (const std::string&): Parameter key
  - `value` (const std::string&): Parameter value
- **Return**: `true` if successful, `false` otherwise
- **Thread Safety**: Protected by mutex

```
bool setValueByPath(const std::string& path, const std::string&
        value);
```

- **Description**: Set value using RFC 6901 JSON Pointer path
- **Parameters**:
  - `path` (const std::string&): Path like "/section/parameter"
  - `value` (const std::string&): New value
- **Return**: `true` if successful, `false` otherwise
- **Note**: Creates intermediate sections if needed
- **Thread Safety**: Protected by mutex

```
bool deleteSection(const std::string& name);
```

- **Description**: Delete entire section and all parameters
- **Parameters**:
  - `name` (const std::string&): Section name
- **Return**: `true` if successful, `false` otherwise
- **Thread Safety**: Protected by mutex

```
bool deleteParameter(const std::string& section, const
        std::string& key);
```

- **Description**: Delete specific parameter
- **Parameters**:
  - `section` (const std::string&): Section name
  - `key` (const std::string&): Parameter key
- **Return**: `true` if successful, `false` otherwise
- **Thread Safety**: Protected by mutex

```
bool deleteByPath(const std::string& path);
```

- **Description**: Delete value using RFC 6901 JSON Pointer path
- **Parameters**:
  - `path` (const std::string&): Path to delete
- **Return**: `true` if successful, `false` otherwise
- **Thread Safety**: Protected by mutex

Query Methods

```
bool sectionExists(const std::string& name) const;
```

- **Description**: Check if section exists
- **Parameters**:
  - `name` (const std::string&): Section name
- **Return**: `true` if exists, `false` otherwise
- **Thread Safety**: Protected by mutex

```
bool parameterExists(const std::string& section, const
        std::string& key) const;
```

- **Description**: Check if parameter exists
- **Parameters**:
  - `section` (const std::string&): Section name
  - `key` (const std::string&): Parameter key
- **Return**: `true` if exists, `false` otherwise
- **Thread Safety**: Protected by mutex

```
std::vector<std::string> getSectionNames() const;
```

- **Description**: Get list of all section names
- **Parameters**: None
- **Return**: Vector of section names
- **Thread Safety**: Protected by mutex

```
std::vector<std::string> getParameterKeys(const std::string&
        section) const;
```

- **Description**: Get list of all parameter keys in section
- **Parameters**:
  - `section` (const std::string&): Section name
- **Return**: Vector of parameter keys
- **Thread Safety**: Protected by mutex

```
size_t getSectionCount() const;
```

- **Description**: Get total number of sections
- **Parameters**: None
- **Return**: Number of sections
- **Thread Safety**: Protected by mutex

```
size_t getParameterCount(const std::string& section) const;
```

- **Description**: Get number of parameters in section
- **Parameters**:
  - `section` (const std::string&): Section name
- **Return**: Number of parameters (0 if section not found)
- **Thread Safety**: Protected by mutex

Merge & Diff Methods

```
bool merge(const OopParser& other,
          MergeStrategy strategy = MergeStrategy::REPLACE);
```

- **Description**: Merge another configuration into this one
- **Parameters**:
    - `other` (const OopParser&): Configuration to merge
    - `strategy` (MergeStrategy): Merge strategy
- **Return**: `true` if successful, `false` otherwise
- **Thread Safety**: Protected by mutex
- **Strategies**:
    - `REPLACE`: Incoming values override
    - `APPEND`: Keep existing, add new
    - `DEEP_MERGE`: Recursive merge
    - `CUSTOM`: User-defined resolver

```
std::vector<DiffEntry> diff(const OopParser& other) const;
```

- **Description**: Find differences between configurations

- **Parameters**:

    - `other` (const OopParser&): Configuration to compare

- **Return**: Vector of differences

- **Thread Safety**: Protected by mutex

- **Example**:

  ```
  auto differences = parser1.diff(parser2);
  for (const auto& diff : differences) {
      std::cout << diff.section << "/" << diff.key <<
           std::endl;
  }
  ```

Error Handling Methods

```
std::string getLastError() const;
```

- **Description**: Get detailed error message from last operation

- **Parameters**: None

- **Return**: Error message string (empty if no error)

- **Example**:

  ```
  if (!parser.loadFromOop("config.oop")) {
      std::cerr << parser.getLastError() << std::endl;
  }
  ```

```
void clearLastError();
```

- **Description**: Clear error message
- **Parameters**: None
- **Return**: void

ConfigBuilder

Builder pattern implementation for fluent configuration construction.

Methods

```
ConfigBuilder& addSection(const std::string& name);
```

- **Description**: Add or select section
- **Parameters**:
    - `name` (const std::string&): Section name
- **Return**: Reference to *this (for chaining)
- **Example**:

  ```
  builder.addSection("object");
  ```

```
ConfigBuilder& addParameter(const std::string& key, const
        std::string& value);
```

- **Description**: Add parameter to current section
- **Parameters**:
    - `key` (const std::string&): Parameter key
    - `value` (const std::string&): Parameter value
- **Return**: Reference to *this (for chaining)
- **Note**: Section must be set first
- **Example**:

  ```
  builder.addParameter("id", "17030").addParameter("name",
          "Vesta");
  ```

```
OopParser build();
```

- **Description**: Build and return OopParser instance
- **Parameters**: None
- **Return**: OopParser with configured data
- **Example**:

```
    auto parser = builder
        .addSection("object")
        .addParameter("id", "123")
        .build();
```

```
ConfigBuilder& reset();
```

- **Description**: Clear all added data
- **Parameters**: None
- **Return**: Reference to *this

VersionedOopParser

Extended OopParser with versioning capabilities.

Methods

```
bool enableVersioning();
```

- **Description**: Enable version history tracking
- **Parameters**: None
- **Return**: `true` if successful
- **Note**: Creates initial snapshot

```
bool createVersion(const std::string& description = "");
```

- **Description**: Create version snapshot with optional description
- **Parameters**:
    - `description` (const std::string&): Version description (optional)
- **Return**: `true` if successful
- **Thread Safety**: Protected by mutex
- **Example**:

```
parser.createVersion("Initial setup");
parser.setParameter("section", "key", "new_value");
parser.createVersion("Updated value");
```

```
bool rollback(size_t versionNumber);
```

- **Description**: Restore configuration to specific version
- **Parameters**:
    - `versionNumber` (size_t): Version to restore (0-based index)
- **Return**: `true` if successful
- **Thread Safety**: Protected by mutex

```
std::vector<VersionEntry> getHistory() const;
```

- **Description**: Retrieve all version history entries

- **Parameters**: None
- **Return**: Vector of VersionEntry structures
- **Structure**:

  ```
  struct VersionEntry {
      size_t number;
      std::string timestamp;
      std::string description;
      std::string data;
  };
  ```

```
size_t getCurrentVersion() const;
```

- **Description**: Get current version number
- **Parameters**: None
- **Return**: Current version index

```
std::string exportVersionsToJson() const;
```

- **Description**: Export version history to JSON format
- **Parameters**: None
- **Return**: JSON string with version information

```
bool deleteOldVersions(size_t keepCount);
```

- **Description**: Delete old versions keeping only recent ones
- **Parameters**:
  - `keepCount` (size_t): Number of versions to keep
- **Return**: `true` if successful
- **Thread Safety**: Protected by mutex

```
bool clearVersioning();
```

- **Description**: Disable versioning and delete all versions
- **Parameters**: None
- **Return**: `true` if successful

BatchProcessor

Process multiple configuration files with statistics.

Methods

```
static BatchStats validateAll(const std::vector<std::string>&
        filepaths);
```

- **Description**: Validate multiple configuration files

- **Parameters**:
  - `filepaths` (const std::vector&): Vector of file paths
- **Return**: BatchStats with results
- **Thread Safety**: Safe for concurrent calls
- **Returns Statistics**:

  ```
  struct BatchStats {
      size_t total_files;
      size_t successful_operations;
      size_t failed_operations;
      std::vector<std::string> failed_files;
      std::vector<std::string> error_messages;
  };
  ```

```
static BatchStats convertAll(const std::vector<std::string>&
        sourceFiles,
                                   const std::string& sourceFormat,
                                   const std::string& targetFormat,
                                   const std::string& outputDirectory
        = "");
```

- **Description**: Convert multiple files to different format
- **Parameters**:
  - `sourceFiles` (const std::vector&): Source file paths
  - `sourceFormat` (const std::string&): Source format (“oop”, “json”, etc.)
  - `targetFormat` (const std::string&): Target format
  - `outputDirectory` (const std::string&): Where to save (optional)
- **Return**: BatchStats with results
- **Example**:

  ```
  auto stats = BatchProcessor::convertAll(
      {"config1.oop", "config2.oop"},
      "oop", "json", "output/"
  );
  ```

```
static BatchStats mergeAll(const std::vector<std::string>&
        filepaths,
                                const std::string& outputFile,
                                MergeStrategy strategy);
```

- **Description**: Merge multiple configuration files
- **Parameters**:
  - `filepaths` (const std::vector&): Files to merge
  - `outputFile` (const std::string&): Output file path
  - `strategy` (MergeStrategy): Merge strategy

- **Return**: BatchStats with results

ConfigSchema

Define and validate configuration structure.

Methods

```
bool loadFromJson(const std::string& filepath);
```

- **Description**: Load schema from JSON file
- **Parameters**:
  - `filepath` (const std::string&): Path to schema JSON file
- **Return**: `true` if successful

```
nlohmann::json toJsonSchema() const;
```

- **Description**: Export schema in JSON Schema format (draft-07)
- **Parameters**: None
- **Return**: JSON Schema representation
- **Standard**: JSON Schema draft-07 compliant

```
bool saveJsonSchema(const std::string& filepath) const;
```

- **Description**: Save schema to JSON Schema file
- **Parameters**:
  - `filepath` (const std::string&): Output file path
- **Return**: `true` if successful

```
bool validateConfiguration(const OopParser& config) const;
```

- **Description**: Validate configuration against schema
- **Parameters**:
  - `config` (const OopParser&): Configuration to validate
- **Return**: `true` if valid

```
std::vector<std::string> getValidationErrors() const;
```

- **Description**: Get list of validation errors from last check
- **Parameters**: None
- **Return**: Vector of error messages

Data Structures

ConfigParameter

Individual configuration parameter.

```
struct ConfigParameter {
    std::string key;              // Parameter name
    std::string value;            // Parameter value
    std::string type;             // Detected type (int, float,
        bool, string)
    std::vector<std::string> tags;// Optional tags
};
```

ConfigSectionData

Configuration section containing parameters.

```
struct ConfigSectionData {
    std::string name;                                 // Section
        name
    std::map<std::string, ConfigParameter>
        params;  // Parameters map
};
```

MergeConflict

Represents conflict during merge operation.

```
struct MergeConflict {
    std::string section;         // Section name
    std::string key;             // Parameter key
    std::string existingValue;   // Current value
    std::string incomingValue;   // Incoming value
};
```

MergeStats

Statistics from merge operation.

```
struct MergeStats {
    size_t sections_merged;      // Number of sections merged
    size_t parameters_merged;    // Number of parameters merged
    size_t conflicts;            // Number of conflicts
    std::vector<MergeConflict> conflictList;
};
```

DiffEntry

Represents difference between configurations.

```
struct DiffEntry {
    std::string section;         // Section name
    std::string key;             // Parameter key
    std::string oldValue;        // Previous value
    std::string newValue;        // New value
    std::string changeType;      // "added", "removed",
        "modified"
};
```

RangeConstraint

Numeric constraint definition.

```
struct RangeConstraint {
    double min;                     // Minimum value
    double max;                     // Maximum value
    bool minInclusive;              // Is minimum inclusive
    bool maxInclusive;              // Is maximum inclusive

    bool validate(double value) const;
};
```

ParameterSpec

Schema specification for a parameter.

```
struct ParameterSpec {
    std::string key;                                // Parameter
        name
    bool required;                                  // Is required
    std::string description;                        // Description
    std::string default_value;                      // Default
        value
    RangeConstraint constraint;                     // Numeric
        constraint
    std::vector<std::string> allowed_values;        // Enum values
};
```

SectionSpec

Schema specification for a section.

```
struct SectionSpec {
    std::string name;                               // Section name
    std::string description;                        // Description
    bool required;                                  // Is required
    std::map<std::string, ParameterSpec> params;    // Parameter
        specs
};
```

ConfigSchema

Schema definition for entire configuration.

```
struct ConfigSchema {
    std::string name;                               // Schema name
    std::string version;                            // Schema
        version
    std::map<std::string, SectionSpec> sections;  // Section
        specs
};
```

VersionEntry

History entry for versioning.

```
struct VersionEntry {
    size_t number;                // Version number
    std::string timestamp;        // Creation timestamp
    std::string description;      // Version description
    std::string data;             // Serialized configuration
};
```

BatchStats

Statistics from batch operations.

```
struct BatchStats {
    size_t total_files;                                 // Total files
        processed
    size_t successful_operations;                       // Successful
        count
    size_t failed_operations;                           // Failed count
    std::vector<std::string> failed_files;              // Failed file
        paths
    std::vector<std::string> error_messages;            // Error
        details
};
```

Enumerations

MergeStrategy

```
enum class MergeStrategy {
    REPLACE,       // Incoming values override existing
    APPEND,        // Keep existing, add new only
    DEEP_MERGE,    // Recursive merge for nested structures
    CUSTOM         // User-defined resolver function
};
```

Usage:

```
parser1.merge(parser2, MergeStrategy::REPLACE);
```

Function Reference

Type Detection

```
std::string OopParser::detectType(const std::string& value);
```

- **Description**: Detect value type
- **Parameters**:
  - `value` (const std::string&): Value to analyze
- **Return**: Type string (“int”, “float”, “bool”, “string”, “json”)
- **Examples**:

```
detectType("123") → "int"
detectType("1.5") → "float"
detectType("true") → "bool"
detectType("hello") → "string"
detectType("{\"a\":1}") → "json"
```

Constants

Format Constants

```
const std::string FORMAT_OOP = "oop";
const std::string FORMAT_JSON = "json";
const std::string FORMAT_XML = "xml";
const std::string FORMAT_CSV = "csv";
const std::string FORMAT_YAML = "yaml";
const std::string FORMAT_TOML = "toml";
```

Type Constants

```
const std::string TYPE_INT = "int";
const std::string TYPE_FLOAT = "float";
const std::string TYPE_BOOL = "bool";
const std::string TYPE_STRING = "string";
const std::string TYPE_JSON = "json";
```

Error Codes

Common Error Messages

Error	Cause	Solution
“File not found”	File path doesn’t exist	Verify file path exists

Error	Cause	Solution
"Invalid format"	File format not recognized	Check format specification
"Permission denied"	Cannot read/write file	Check file permissions
"Section not found"	Referenced section missing	Verify section name
"Parameter not found"	Referenced parameter missing	Verify parameter key
"Invalid JSON"	Malformed JSON content	Validate JSON syntax
"Parsing error"	Format parser error	Check file format compliance
"YAML not supported"	YAML support not compiled	Recompile with -DWITH_YAML
"TOML not supported"	TOML support not compiled	Recompile with -DWITH_TOML

Type Conversions

Automatic Type Detection

Value	Detected Type	Conversion
"123"	int	`std::stoi("123")`
"-456"	int	`std::stoi("-456")`
"1.5"	float	`std::stof("1.5")`
"-2.3"	float	`std::stof("-2.3")`
"true", "false"	bool	Literal match
"{...}"	json	`nlohmann::json::parse()`
Everything else	string	As-is

Manual Type Conversion

```
// String to int
ConfigParameter* param = parser.getParameter("section", "count");
int value = std::stoi(param->value);

// String to float
double value = std::stod(param->value);

// String to bool
bool value = (param->value == "true");

// String to JSON
auto json = nlohmann::json::parse(param->value);
```

Complete Example

```
#include <ioc_config/oop_parser.h>
#include <iostream>
```

```cpp
using namespace ioc_config;

int main() {
    // 1. Create parser
    OopParser parser;

    // 2. Load configuration
    if (!parser.loadFromOop("config.oop")) {
        std::cerr << "Error: " << parser.getLastError() <<
         std::endl;
        return 1;
    }

    // 3. Access data
    std::string name = parser.getParameter("object", "name",
         "Unknown");
    std::cout << "Object name: " << name << std::endl;

    // 4. Modify data
    parser.setParameter("object", "discovered", "2023-01-15");

    // 5. Save to different format
    parser.saveToJson("config.json");

    // 6. Create version
    VersionedOopParser vparser;
    vparser.enableVersioning();
    vparser.setParameter("object", "status", "updated");
    vparser.createVersion("Status updated");

    // 7. Merge configurations
    OopParser other;
    other.loadFromOop("other_config.oop");
    parser.merge(other, MergeStrategy::APPEND);

    return 0;
}
```

See Also

- **ARCHITECTURE.md**: System design and patterns
- **USAGE_GUIDE.md**: Practical usage examples
- **IMPLEMENTATION_GUIDE.md**: Integration with projects