cppSwitchboard API Reference

Complete API Documentation

cppSwitchboard Development Team

June 14, 2025

Contents

1 cppSwitchboard API Reference **1**
- 1.1 Table of Contents . . . 1
- 1.2 Overview . . . 2
- 1.3 Core Classes . . . 2
- 1.4 HTTP Request and Response . . . 2
- 1.5 Routing System . . . 3
- 1.6 Configuration Management . . . 4
- 1.7 Debugging and Logging . . . 5
- 1.8 HTTP/2 Support . . . 5
- 1.9 Usage Examples . . . 6
- 1.10 Error Handling . . . 7
- 1.11 Performance and Best Practices . . . 8
- 1.12 Building and Integration . . . 8

1 cppSwitchboard API Reference

Version: 1.0.0
Modern C++ HTTP/1.1 and HTTP/2 Server Library

1.1 Table of Contents

1. Overview
2. Core Classes
3. HTTP Server
4. HTTP Request and Response
5. Routing System
6. Configuration Management
7. Debugging and Logging
8. HTTP/2 Support
9. Usage Examples
10. Error Handling

1.2 Overview

The cppSwitchboard library provides a modern, high-performance HTTP server implementation in C++ supporting both HTTP/1.1 and HTTP/2 protocols. It features a flexible routing system, comprehensive configuration management, and built-in debugging capabilities.

1.2.1 Key Features

- **Dual Protocol Support**: Both HTTP/1.1 and HTTP/2
- **Modern C++**: Built with C++17 standards
- **High Performance**: Asynchronous request handling
- **Flexible Routing**: Pattern-based URL routing with parameter extraction
- **Configuration Driven**: YAML-based configuration with validation
- **Comprehensive Testing**: 100% test coverage
- **Debug Support**: Built-in logging and debugging utilities

---

1.3 Core Classes

1.3.1 HttpServer

The main server class that handles incoming HTTP connections and routes requests.

Constructor:

`HttpServer(const ServerConfig& config)`

Key Methods: - `void start()` - Starts the HTTP server - `void stop()` - Gracefully stops the server - `void registerRoute(const std::string& pattern, HttpMethod method, HttpHandler handler)` - Registers a route handler - `bool isRunning() const` - Checks if server is currently running

Example Usage:

```
#include <cppSwitchboard/http_server.h>

ServerConfig config;
config.http1.port = 8080;
config.http1.bindAddress = "0.0.0.0";

HttpServer server(config);

server.registerRoute("/api/users", HttpMethod::GET, [](const HttpRequest& req) {
    return HttpResponse::json("{\"users\": []}");
});

server.start();
```

1.4 HTTP Request and Response

1.4.1 HttpRequest Class

Represents an incoming HTTP request with all its components.

Properties: - `std::string getMethod() const` - HTTP method (GET, POST, etc.) - `std::string getPath() const` - Request path - `std::string getQuery() const` - Query string - `std::string getHeader(const std::string& name) const` - Get header value - `std::string getBody() const` - Request body - `std::string getQueryParam(const std::string& name) const` - Get query parameter

Methods: - `void parseQueryString(const std::string& query)` - Parse query parameters - `void addHeader(const std::string& name, const std::string& value)` - Add header - `bool hasHeader(const std::string& name) const` - Check if header exists

1.4.2 HttpResponse Class

Represents an HTTP response to be sent back to the client.

Constructor:

```
HttpResponse(int status = 200, const std::string& body = "")
```

Static Factory Methods: - `static HttpResponse ok(const std::string& body)` - 200 OK response - `static HttpResponse json(const std::string& body)` - JSON response with correct headers - `static HttpResponse html(const std::string& body)` - HTML response with correct headers - `static HttpResponse notFound()` - 404 Not Found response - `static HttpResponse internalError()` - 500 Internal Server Error response

Methods: - `void setStatus(int status)` - Set HTTP status code - `void setBody(const std::string& body)` - Set response body - `void addHeader(const std::string& name, const std::string& value)` - Add header - `int getStatus() const` - Get status code - `std::string getBody() const` - Get response body - `std::string getContentType() const` - Get content type header

Example Usage:

```
// Simple text response
auto response = HttpResponse::ok("Hello, World!");

// JSON response
auto jsonResponse = HttpResponse::json("{\"message\": \"Success\"}");

// Custom response
HttpResponse custom(201);
custom.setBody("Created");
custom.addHeader("Location", "/api/users/123");
```

1.5 Routing System

1.5.1 RouteRegistry Class

Manages URL patterns and route matching for the HTTP server.

Methods: - `void registerRoute(const std::string& pattern, HttpMethod method, HttpHandler handler)` - Register a route - `RouteMatch findRoute(const std::string& path, HttpMethod method)` - Find matching route - `RouteMatch findRoute(const HttpRequest& request)` - Find route from request - `void clearRoutes()` - Remove all registered routes

1.5.2 Route Patterns

The routing system supports flexible URL patterns:

Static Routes:

```
server.registerRoute("/api/users", HttpMethod::GET, handler);
```

Parameterized Routes:

```
server.registerRoute("/api/users/{id}", HttpMethod::GET, handler);
server.registerRoute("/api/users/{id}/posts/{postId}", HttpMethod::GET, handler);
```

HTTP Methods: - `HttpMethod::GET` - `HttpMethod::POST` - `HttpMethod::PUT` - `HttpMethod::DELETE` - `HttpMethod::PATCH` - `HttpMethod::HEAD` - `HttpMethod::OPTIONS`

1.5.3 Handler Functions

Route handlers can be defined as lambda functions or function pointers:

```
// Lambda handler
server.registerRoute("/hello", HttpMethod::GET, [](const HttpRequest& req) {
    return HttpResponse::ok("Hello, " + req.getQueryParam("name"));
});

// Function handler
HttpResponse userHandler(const HttpRequest& request) {
    return HttpResponse::json("{\"user\": \"data\"}");
}
server.registerRoute("/user", HttpMethod::GET, userHandler);
```

1.6 Configuration Management

1.6.1 ServerConfig Structure

The main configuration structure that defines server behavior:

```
struct ServerConfig {
    ApplicationConfig application;
    Http1Config http1;
    Http2Config http2;
    SslConfig ssl;
    DebugLoggingConfig debug;
    SecurityConfig security;
    MonitoringConfig monitoring;
};
```

1.6.2 Configuration Loading

ConfigLoader Class: - `static std::unique_ptr<ServerConfig> loadFromFile(const std::string& filename)` - Load from YAML file - `static std::unique_ptr<ServerConfig> loadDefaults()` - Load default configuration - `static bool validateConfig(const ServerConfig& config)` - Validate configuration

Example Configuration (YAML):

```
application:
  name: "My HTTP Server"
  version: "1.0.0"
  environment: "development"

http1:
  enabled: true
  port: 8080
  bindAddress: "0.0.0.0"

http2:
  enabled: true
  port: 8443
```

```
  bindAddress: "0.0.0.0"

ssl:
  enabled: true
  certificateFile: "/path/to/cert.pem"
  privateKeyFile: "/path/to/key.pem"

debug:
  enabled: true
  logLevel: "info"
  logFile: "/var/log/server.log"
```

1.6.3 Configuration Validation

ConfigValidator Class: - `static bool validateConfig(const ServerConfig& config)` - Validate entire configuration - `static bool validatePorts(const ServerConfig& config)` - Validate port configurations - `static bool validateSsl(const ServerConfig& config)` - Validate SSL settings

---

1.7 Debugging and Logging

1.7.1 DebugLogger Class

Provides comprehensive logging capabilities for debugging and monitoring.

Constructor:

`DebugLogger(const DebugLoggingConfig& config)`

Methods: - `void info(const std::string& message)` - Log info message - `void warn(const std::string& message)` - Log warning message - `void error(const std::string& message)` - Log error message - `void debug(const std::string& message)` - Log debug message - `void setLogLevel(const std::string& level)` - Set logging level

Usage Example:

```
DebugLoggingConfig logConfig;
logConfig.enabled = true;
logConfig.logLevel = "debug";
logConfig.logFile = "/var/log/server.log";

DebugLogger logger(logConfig);
logger.info("Server starting...");
logger.debug("Processing request: " + request.getPath());
```

1.8 HTTP/2 Support

1.8.1 Http2Server Class

Dedicated HTTP/2 server implementation with advanced features.

Key Features: - Stream multiplexing - Header compression (HPACK) - Server push capabilities - Flow control

Configuration:

```
Http2Config config;
config.enabled = true;
config.port = 8443;
config.maxConcurrentStreams = 100;
config.initialWindowSize = 65535;
```

1.9 Usage Examples

1.9.1 Basic HTTP Server

```
#include <cppSwitchboard/http_server.h>
#include <cppSwitchboard/config.h>

int main() {
    // Load configuration
    auto config = ConfigLoader::loadDefaults();
    config->http1.port = 8080;

    // Create server
    HttpServer server(*config);

    // Register routes
    server.registerRoute("/", HttpMethod::GET, [](const HttpRequest& req) {
        return HttpResponse::html("<h1>Welcome to cppSwitchboard!</h1>");
    });

    server.registerRoute("/api/status", HttpMethod::GET, [](const HttpRequest& req) {
        return HttpResponse::json("{\"status\": \"ok\", \"uptime\": 12345}");
    });

    // Start server
    server.start();

    return 0;
}
```

1.9.2 RESTful API Example

```
// GET /api/users
server.registerRoute("/api/users", HttpMethod::GET, [](const HttpRequest& req) {
    // Return list of users
    return HttpResponse::json("[{\"id\": 1, \"name\": \"John\"}]");
});

// GET /api/users/{id}
server.registerRoute("/api/users/{id}", HttpMethod::GET, [](const HttpRequest& req) {
    std::string userId = req.getPathParam("id");
    return HttpResponse::json("{\"id\": " + userId + ", \"name\": \"John\"}");
});

// POST /api/users
server.registerRoute("/api/users", HttpMethod::POST, [](const HttpRequest& req) {
    std::string body = req.getBody();
```

```
    // Process user creation
    return HttpResponse(201, "{\"id\": 123, \"created\": true}");
});

// PUT /api/users/{id}
server.registerRoute("/api/users/{id}", HttpMethod::PUT, [](const HttpRequest& req) {
    std::string userId = req.getPathParam("id");
    std::string body = req.getBody();
    // Process user update
    return HttpResponse::ok("{\"updated\": true}");
});

// DELETE /api/users/{id}
server.registerRoute("/api/users/{id}", HttpMethod::DELETE, [](const HttpRequest& req) {
    std::string userId = req.getPathParam("id");
    // Process user deletion
    return HttpResponse(204); // No content
});
```

1.9.3 Middleware Example

```
// Custom middleware for authentication
auto authMiddleware = [](const HttpRequest& req) -> bool {
    std::string token = req.getHeader("Authorization");
    return !token.empty() && token.substr(0, 7) == "Bearer ";
};

// Protected route
server.registerRoute("/api/protected", HttpMethod::GET, [authMiddleware](const HttpRequest& req) {
    if (!authMiddleware(req)) {
        return HttpResponse(401, "{\"error\": \"Unauthorized\"}");
    }
    return HttpResponse::json("{\"data\": \"secret\"}");
});
```

1.10 Error Handling

1.10.1 Exception Types

The library defines several exception types for different error conditions:

- `ConfigurationException` - Configuration-related errors
- `NetworkException` - Network and connection errors
- `RoutingException` - Route registration and matching errors
- `HttpException` - HTTP protocol errors

1.10.2 Error Response Helpers

```
// Standard error responses
auto notFound = HttpResponse::notFound(); // 404
auto serverError = HttpResponse::internalError(); // 500

// Custom error responses
HttpResponse badRequest(400, "{\"error\": \"Invalid request\"}");
```

```
HttpResponse unauthorized(401, "{\"error\": \"Authentication required\"}");
HttpResponse forbidden(403, "{\"error\": \"Access denied\"}");
```

1.10.3 Error Handling Best Practices

```
server.registerRoute("/api/data", HttpMethod::GET, [](const HttpRequest& req) {
    try {
        // Process request
        std::string data = processData(req);
        return HttpResponse::json(data);
    } catch (const std::invalid_argument& e) {
        return HttpResponse(400, "{\"error\": \"" + std::string(e.what()) + "\"}");
    } catch (const std::exception& e) {
        // Log error
        logger.error("Unexpected error: " + std::string(e.what()));
        return HttpResponse::internalError();
    }
});
```

1.11 Performance and Best Practices

1.11.1 Threading Model

- The server uses an asynchronous, event-driven architecture
- Request handlers should be thread-safe
- Avoid blocking operations in handlers

1.11.2 Memory Management

- Use RAII principles for resource management
- Prefer smart pointers for dynamic allocation
- Be mindful of request/response object lifetimes

1.11.3 Configuration Optimization

```
// Production configuration example
Http1Config prodConfig;
prodConfig.maxConnections = 1000;
prodConfig.keepAliveTimeout = 5;
prodConfig.maxRequestSize = 1024 * 1024; // 1MB

Http2Config http2Config;
http2Config.maxConcurrentStreams = 200;
http2Config.initialWindowSize = 65535;
```

1.12 Building and Integration

1.12.1 CMake Integration

```
find_package(cppSwitchboard REQUIRED)

target_link_libraries(your_target
```

```
    PRIVATE cppSwitchboard::cppSwitchboard
)
```

1.12.2 Dependencies

- C++17 compatible compiler
- OpenSSL (for HTTPS/HTTP2 support)
- CMake 3.15+

This API reference provides comprehensive documentation for the cppSwitchboard library. For additional examples and detailed usage patterns, refer to the examples directory and test suite.