17.2 ConfigurationAdmin .................................................. 211
17.3 ConfigurationException ........................................... 213
17.4 ConfigurationListener ............................................... 213
17.5 ManagedService .................................................... 215
17.6 ManagedServiceFactory .......................................... 215

18 LogService API .......................................................... 217
18.1 LogService ............................................................. 217

19 AsyncWorkService API ................................................ 219
19.1 AsyncWorkService .................................................. 219

20 Http Service API ........................................................ 221
20.1 HttpServlet .......................................................... 221
20.2 HttpServletRequest .................................................. 223
20.3 HttpServletResponse ............................................... 224
20.4 ServletConfig ........................................................ 227
20.5 ServletContainer .................................................... 228
20.6 ServletContext ....................................................... 229

21 Web Console API ........................................................ 231
21.1 AbstractWebConsolePlugin ....................................... 231
21.2 SimpleWebConsolePlugin ......................................... 234
21.3 WebConsoleConstants .............................................. 235
21.4 WebConsoleDefaultVariableResolver ......................... 237
21.5 WebConsoleVariableResolver ................................... 237

22 Shell Service API ........................................................ 239
22.1 ShellService ........................................................... 239

23 Change Log ............................................................. 241
23.1 (UNRELEASED) vX.X.X (20XX-XX-XX) ......................... 241
23.2 Added ................................................................. 241
23.3 Changed ............................................................. 241
23.4 Removed ............................................................. 241
23.5 Deprecated .......................................................... 241
23.6 Fixed ................................................................. 241
23.7 v3.7.2 (2022-06-16) ................................................ 241
23.8 General Note ......................................................... 242
23.9 Added ................................................................. 242
23.10 Changed ............................................................ 242
23.11 Removed ............................................................ 242
23.12 Deprecated ........................................................ 243
23.13 Fixed ................................................................. 243
23.14 v3.6.0 (2020-08-13) ................................................ 243
23.15 Added ............................................................... 244
23.16 Changed ........................................................... 244
23.17 Removed ............................................................ 244
23.18 Deprecated ........................................................ 244
23.19 Fixed ................................................................. 244
23.20 v3.5.0 (2020-07-04) ................................................ 244
23.21 Added ............................................................... 244
23.22 Changed ........................................................... 244
23.23 Removed ............................................................ 245
23.24 Deprecated ........................................................ 245
23.25 Fixed ....................................................... 245
23.26 v3.4.0 (2019-12-10) ........................................... 245
23.27 Added ....................................................... 245
23.28 Changed .................................................... 246
23.29 Removed .................................................... 246
23.30 Deprecated ................................................. 246
23.31 Fixed ....................................................... 246
23.32 v3.3.0 (2018-02-20) ........................................... 247
23.33 Added ....................................................... 247
23.34 Fixed ....................................................... 247
23.35 v3.2.0 (2017-10-30) ........................................... 247
23.36 Added ....................................................... 247
23.37 Changed .................................................... 247
23.38 Fixed ....................................................... 247
23.39 v3.1.0 (2017-06-01) ........................................... 248
23.40 v3.0.0 (2017-02-08) ........................................... 249
23.41 v2.1.1 (2014-01-22) ........................................... 250
23.42 v2.1.0 (2014-01-11) ........................................... 250
23.43 v2.0.0 (2013-12-23) ........................................... 250
23.44 v1.0.0 (2013-07-18) ........................................... 251
The CppMicroServices project is a C++ library for creating and managing modular software systems based on ideas from OSGi. The code is open source, and available on github.

The main documentation for the site is organized into a couple sections. Start reading at the top for a quick start.

### Continuous Integration Status

<table>
<thead>
<tr>
<th>Branch</th>
<th>GCC 7.5.0 and 9.4.0</th>
<th>Visual Studio 2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang 9.0</td>
<td>Visual Studio 2022</td>
<td></td>
</tr>
<tr>
<td>Xcode 13.2</td>
<td>MinGW-w64</td>
<td></td>
</tr>
<tr>
<td>Xcode 13.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>master</td>
<td></td>
<td></td>
</tr>
<tr>
<td>development</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.1 Introduction

The C++ Micro Services project is a collection of components for building modular and dynamic service-oriented applications. It is based on OSGi, but tailored to support native cross-platform solutions.

Proper usage of C++ Micro Services patterns and concepts leads to systems with one or more of the following properties:

• Re-use of software components
• Loose coupling between service providers and consumers
• Separation of concerns, based on a service-oriented design
• Clean APIs based on service interfaces
• Extensible and reconfigurable systems

1.2 Requirements

None, except a recent enough C++ compiler. All third-party library dependencies are included and mostly used for implementation details.

1.3 Supported Platforms

The library makes use of C++17 language and library features and compiles on many different platforms.

Recommended absolute minimum required compiler versions:
• GCC 7.5.0
• Clang 9.0
• Clang from Xcode 10.0 (not tested)
• Visual Studio 2017 (MSVC++ 15.0) (not tested)

Not all of the absolute minimum compiler versions are tested (as noted). We test and recommend the following compilers:
• GCC 9.4.0
• Clang 9.0
• Clang from Xcode 13.2 and 13.4
• Visual Studio 2019 and 2022

Recommended minimum required CMake version:
• CMake 3.17.0

For all CI builds through GitHub Actions, the CMake version (and version of other provided software) we use is determined by the software provided on the GitHub-hosted runners.

For information about the specific versions of software the runners use, please see the following resources:
• macOS-12 Runner Information <https://github.com/actions/virtual-environments/blob/main/images/macos/macos-12-Readme.md>

Below is a list of tested compiler/OS combinations:
• GCC 7.5.0 (Ubuntu 18.04)
• GCC 9.4.0 (Ubuntu 20.04)
• Clang 9.0 (Ubuntu 18.04)
• Apple Clang, Xcode 13.2.0 (OS X 11.6.8)
• Apple Clang, Xcode 13.4.0 (OS X 12.5.0)
• Visual Studio 2019
• Visual Studio 2022
• MinGW-w64
1.4 Legal

The C++ Micro Services project was initially developed at the German Cancer Research Center. Its source code is hosted as a GitHub Project. See the COPYRIGHT file in the top-level directory for detailed copyright information.

This project is licensed under the Apache License v2.0.

1.5 Code of Conduct

CppMicroServices.org welcomes developers with different backgrounds and a broad range of experience. A diverse and inclusive community will create more great ideas, provide more unique perspectives, and produce more outstanding code. Our aim is to make the CppMicroServices community welcoming to everyone.

To provide clarity of what is expected of our members, CppMicroServices has adopted the code of conduct defined by contributor-covenant.org. This document is used across many open source communities, and we believe it articulates our values well.

Please refer to the Code of Conduct for further details.

1.6 Quick Start

Start by cloning the project repository. It is important to note that since the project utilizes git submodules, you must clone the repository with the --recursive flag. This will also clone the submodules and place them in their respective directories. For further reading about how git submodules work and how to clone them into an already existing repository on your disk, please see Git’s documentation.

Essentially, the C++ Micro Services library provides you with a powerful dynamic service registry on top of a managed lifecycle. The framework manages, among other things, logical units of modularity called bundles that are contained in shared or static libraries. Each bundle within a library has an associated cppmicroservices::BundleContext object, through which the service registry is accessed.

To query the registry for a service object implementing one or more specific interfaces, the code would look like this:

```cpp
#include "cppmicroservices/BundleContext.h"
#include "SomeInterface.h"

using namespace cppmicroservices;

void UseService(BundleContext context)
{
  auto serviceRef = context.GetServiceReference<SomeInterface>();
  if (serviceRef)
  {
    auto service = context.GetService(serviceRef);
    if (service) /* do something */
  }
}
```

Registering a service object against a certain interface looks like this:

```cpp
#include "cppmicroservices/BundleContext.h"
#include "SomeInterface.h"

using namespace cppmicroservices;
```
The OSGi service model additionally allows to annotate services with properties and using these properties during service look-ups. It also allows to track the life-cycle of service objects. Please see the Documentation for more examples and tutorials and the API reference. There is also a blog post about OSGi Lite for C++.

### 1.7 Git Branch Conventions

The Git repository contains two eternal branches, `master` and `development`. The master branch contains production quality code and its HEAD points to the latest released version. The development branch is the default branch and contains the current state of development. Pull requests by default target the development branch. See the `CONTRIBUTING` file for details about the contribution process.
The C++ Micro Services library provides CMake build scripts which allow the generation of platform and IDE specific project or Make files.

### 2.1 Prerequisites

- CMake 3.12.4 (users of the latest Visual Studio should typically also use the latest CMake version available)

### 2.2 Configuration

When building the C++ Micro Services project, you have a few configuration options at hand.

- **CMAKE_INSTALL_PREFIX** The installation path.
- **US_ENABLE_THREADING_SUPPORT** Enable the use of synchronization primitives (atomics and pthread mutexes or Windows primitives) to make the API thread-safe. All documented guarantees of thread-safety are valid if and only if this option is turned ON. If your application is not multi-threaded, turn this option OFF to get maximum performance.

  **Note:** In version 3.0 and 3.1 this option only supported the ON value. The OFF configuration is supported again in version 3.2 and later.

- **BUILD_SHARED_LIBS** Specify if the library should be build shared or static. See *Static Bundles* for detailed information about static CppMicroServices bundles.
- **US_BUILD_TESTING** Build unit tests and code snippets.
- **US_BUILD_EXAMPLES** Build the tutorial code and other examples.
- **US_BUILD_DOC_HTML** Build the html documentation, as seen on docs.cppmicroservices.org.
- **US_BUILD_DOC_MAN** Build the man pages. This is typically only enabled on a Unix-like system.
• **US_ENABLE_ASAN** Enable the use of Address Sanitizer in builds of CppMicroServices.

**Note:** When this setting is enabled for a CppMicroServices build on Windows, the US_ASAN_USER_DLL needs to be correctly set. This DLL is typically found at the following location (VS is the root Visual Studio install folder): $VS\2019\Community\VC\Tools\MSVC\<version>\bin\Hostx64\x64\clang_rt.asan_dbg_dynamic-x86_64.dll.

Additionally, ASAN for Visual Studio is only available in versions 16.9 or later. For more information, please see the Microsoft documentation [https://docs.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-160](https://docs.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-160).

• **US_ASAN_USER_DLL** The path to the ASAN DLL which Microsoft Visual Studio ships. This value should only be set on Windows.

• **US_ENABLE_TSAN** Enable the use of Thread Sanitizer in builds of CppMicroServices. This boolean should only be set on Linux or Mac.

• **US_USE_SYSTEM_GTEST** Build using an existing installation of Google Test.

• **GTEST_ROOT** Specify the root directory of the Google Test framework installation to use when building and running tests.

**Note:** Building the documentation requires

- a Python installation,
- the Doxygen tool,
- the Sphinx documentation generator,
- and the breathe and Read the Docs Sphinx Theme Sphinx extensions

After installing Python and Doxygen, the remaining dependencies can be installed using pip:

```bash
python -m pip install sphinx breathe sphinx_rtd_theme
```

2.3 Building

After configuring a build directory with CMake, the project can be built. If you chose e.g. Unix Makefiles, just type:

```bash
make -j
```

in a terminal with the current directory set to the build directory. To install the libraries, header files, and documentation into the configured CMAKE_INSTALL_PREFIX type:

```bash
make install
```

2.4 Integration

Projects using the C++ Micro Services library need to set-up the correct include paths and link dependencies. Further, each executable or library which needs a BundleContext must contain specific initialization code, a manifest.json resource, and must be compiled with a unique US_BUNDLE_NAME pre-processor definition. See [Getting Started](#) for an introduction to the basic concepts.
The C++ Micro Services library provides CMake Support for CMake based projects but there are no restrictions on the type of build system used for a project.

### 2.4.1 CMake based projects

To easily set-up include paths and linker dependencies, use the common find_package mechanism provided by CMake:

```cmake
project(CppMicroServicesExamples)
set(CMAKE_CXX_STANDARD_REQUIRED 1)
set(CMAKE_CXX_STANDARD 17)
find_package(CppMicroServices NO_MODULE REQUIRED)
cmake_minimum_required(VERSION ${US_CMAKE_MINIMUM_REQUIRED_VERSION})
cmake_policy(VERSION ${US_CMAKE_MINIMUM_REQUIRED_VERSION})
```

The CMake code above sets up a basic project (called `CppMicroServicesExamples`) and tries to find the CppMicroServices package and subsequently to set the necessary include directories. Building a shared library might then look like this:

```cmake
# The library name for the bundle
set(_lib_name dictionaryservice)

# A list of source code files
set(_srcs
  Activator.cpp
  IDictionaryService.cpp
)

# Add a special source file to the _srcs variable that
# will enable dependency checking on bundle resources.
usFunctionGetResourceSource(TARGET Tutorial-$({_lib_name}) OUT _srcs)

# Generate bundle initialization code
usFunctionGenerateBundleInit(TARGET Tutorial-$({_lib_name}) OUT _srcs)

# Create the library
add_library(Tutorial-$({_lib_name}) ${_srcs})

# Add the US_BUNDLE_NAME target property
set_property(TARGET Tutorial-$({_lib_name}) APPEND PROPERTY US_BUNDLE_NAME ${_lib_name})

# Add the required compile definition
set_property(TARGET Tutorial-$({_lib_name}) APPEND PROPERTY COMPILE_DEFINITIONS US_-
˓→BUNDLE_NAME=${_lib_name})

# Embed the manifest file
usFunctionEmbedResources(TARGET Tutorial-$({_lib_name}) WORKING_DIRECTORY resources-
˓→FILES manifest.json)

# Link the CppMicroServices library
target_link_libraries(Tutorial-$({_lib_name}) ${CppMicroServices_LIBRARIES})
```

The call to `usFunctionGenerateBundleInit` generates the proper bundle initialization code and provides access to the bundle specific BundleContext instance. Further, the `set_property` command sets the
US_BUNDLE_NAME definition.

## 2.4.2 Makefile based projects

The following Makefile is located at `/doc/src/examples/makefile/Makefile` and demonstrates a minimal build script:

```makefile
CXX = $(US_CXX)
CXXFLAGS = -g -Wall -Wno-unused -pedantic -fPIC $(US_CXX_FLAGS)
LDFLAGS = -Wl,-rpath="$(CppMicroServices_ROOT)/lib" -Wl,-rpath=.
LDLIBS = "$\{US_CPPMICROSERVICES_TARGET\}"
INCLUDEDIRS = -I"$(CppMicroServices_ROOT)/include/cppmicroservices3"
LIBDIRS = -L"$(CppMicroServices_ROOT)/lib" -L.
RC = "$(CppMicroServices_ROOT)/bin/usResourceCompiler3"

OBJECTS = bundle.o IDictionaryService.o

all : main libbundle.so

main: libbundle.so main.o

libbundle.so: $(OBJECTS) resources.zip

main.o: main.cpp $(CXX) $(CXXFLAGS) -DUS_BUNDLE_NAME=main $(INCLUDEDIRS) -c ../../main.cpp -o main.o

%.o: %.cpp $(CXX) $(CXXFLAGS) -DUS_BUNDLE_NAME=main $(INCLUDEDIRS) -c $< -o $@

resources.zip: resources/manifest.json
    $(RC) -m $< -n bundle -o resources.zip

.PHONY : clean
clean:
    rm -f *.o

The variable `CppMicroServices_ROOT` is an environment variable and must be set to the CppMicroServices installation directory prior to invoking `make`. The bundle initialization code for the `libbundle.so` shared library is generated by using the `CPPMICROSERVICES_INITIALIZE_BUNDLE` pre-processor macro at the end of the `bundle.cpp` source file (any source file compiled into the bundle would do):

```c
#include "cppmicroservices/BundleInitialization.h"
CPPMICROSERVICES_INITIALIZE_BUNDLE
```

See also:

See the *Getting Started* section and the general *Framework* documentation to learn more about fundamental concepts.
This section gives you a quick tour of the most important concepts in order to get you started with developing bundles and publishing and consuming services.

We will define a service interface, service implementation, and service consumer. A simple executable then shows how to install and start these bundles.

The complete source code for this example can be found at /doc/src/examples/getting_started. It works for both shared and static builds, but only the more commonly used shared build mode is discussed in detail below.

## 3.1 The Build System

These examples come with a complete CMakeLists.txt file showing the main usage scenarios of the provided CMake helper functions. The script requires the CppMicroServices package:

```cmake
project(CppMicroServicesGettingStarted)
find_package(CppMicroServices REQUIRED)
```

## 3.2 A Simple Service Interface

Services implement one or more service interfaces. An interface can be any C++ class, but typically contains only pure virtual functions. For our example, we create a separate library containing a service interface that allows us to retrieve the number of elapsed milliseconds since the POSIX epoch:

```cpp
#include <servicetime_export.h>
#include <chrono>

struct SERVICETIME_EXPORT ServiceTime
```
virtual ~ServiceTime();

// Return the number of milliseconds since POSIX epoch time
virtual std::chrono::milliseconds elapsed() const = 0;
};

The CMake code does not require C++ Micro Services specific additions:

```cmake
# A library providing the ServiceTime interface
#
# Technically, this is not a bundle.

add_library(ServiceTime
  service_time/ServiceTime.h
  service_time/ServiceTime.cpp
)

target_include_directories(ServiceTime PUBLIC
  ${BUILD_INTERFACE}:${CMAKE_CURRENT_SOURCE_DIR}/service_time
  ${BUILD_INTERFACE}:${CMAKE_CURRENT_BINARY_DIR}
)
generate_export_header(ServiceTime)

if(BUILD_SHARED_LIBS)
  set_target_properties(ServiceTime PROPERTIES
    CXX_VISIBILITY_PRESET hidden
    VISIBILITY_INLINES_HIDDEN 1
  )
endif()
```

### 3.3 Bundle and BundleContext

A **bundle** is the logical set of C++ Micro Services specific initialization code, metadata stored in a `manifest.json` file, and other resources and code. Multiple bundles can be part of the same or different (shared or static) library or executable. To create the bundle initialization code, you can either use the `usFunctionGenerateBundleInit` CMake function or the `CPPMICROSERVICES_INITIALIZE_BUNDLE` macro directly.

In order to publish and consume a service, we need a **BundleContext** instance, through which a bundle accesses the C++ Micro Services API. Each bundle is associated with a distinct bundle context that is accessible from anywhere in the bundle via the `GetBundleContext()` function:

```cpp
#include <cppmicroservices/GetBundleContext.h>

void Dummy()
{
  auto context = cppmicroservices::GetBundleContext();
}
```

Please note that trying to use `GetBundleContext()` without proper initialization code in the using library will lead to compile or runtime errors.
3.4 Publishing a Service

Publishing a service is done by calling the `BundleContext::RegisterService` function. The following code for the `service_time_systemclock` bundle implements the `ServiceTime` interface as a service:

```cpp
#include <ServiceTime.h>
#include <cppmicroservices/BundleActivator.h>
using namespace cppmicroservices;

class ServiceTimeSystemClock : public ServiceTime
{
    std::chrono::milliseconds elapsed() const
    {
        auto now = std::chrono::system_clock::now();
        // Relies on the de-facto standard of relying on
        // POSIX time in all known implementations so far.
        return std::chrono::duration_cast<std::chrono::milliseconds>(
            now.time_since_epoch());
    }
};

class ServiceTimeActivator : public BundleActivator
{
    void Start(BundleContext ctx)
    {
        auto service = std::make_shared<ServiceTimeSystemClock>();
        ctx.RegisterService<ServiceTime>(service);
    }

    void Stop(BundleContext)
    {
        // Nothing to do
    }
};
CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(ServiceTimeActivator)
```

A `std::shared_ptr` holding the service object is passed as the an argument to the `RegisterService<>()` function within a `bundle activator`. The service is registered as long as it is explicitly un-registered or the bundle is stopped. The bundle activator is optional, but if it is declared, its `BundleActivator::Start(BundleContext)` and `BundleActivator::Stop(BundleContext)` functions are called when the bundle is `started` or `stopped`, respectively.

The CMake code for creating our bundle looks like this:

```cmake
# A bundle implementing the ServiceTime interface
set(_srcs
    service_time_systemclock/ServiceTimeImpl.cpp
)
# Set up dependencies to resources to track changes
usFunctionGetResourceSource(TARGET ServiceTime_SystemClock OUT _srcs)
```
# Generate bundle initialization code
usFunctionGenerateBundleInit(TARGET ServiceTime_SystemClock OUT _srcs)

add_library(ServiceTime_SystemClock ${_srcs})

target_link_libraries(ServiceTime_SystemClock CppMicroServices ServiceTime)

set(_bundle_name service_time_systemclock)

set_target_properties(ServiceTime_SystemClock PROPERTIES
  # This is required for every bundle
  COMPILE_DEFINITIONS US_BUNDLE_NAME=${_bundle_name}
  # This is for convenience, used by other CMake functions
  US_BUNDLE_NAME ${_bundle_name}
)

if(BUILD_SHARED_LIBS)
  set_target_properties(ServiceTime_SystemClock PROPERTIES
    CXX_VISIBILITY_PRESET hidden
    VISIBILITY_INLINES_HIDDEN 1
  )
endif()

# Embed meta-data from a manifest.json file
usFunctionEmbedResources(TARGET ServiceTime_SystemClock
  WORKING_DIRECTORY
    ${CMAKE_CURRENT_SOURCE_DIR}/service_time_systemclock
  FILES
    manifest.json
)

In addition to the generated bundle initialization code, we need to specify a unique bundle name by using the US_BUNDLE_NAME compile definition as shown above.

We also need to provide the `manifest.json` file, which is added as a resource and contains the following JSON data:

```json
{
    "bundle.symbolic_name" : "service_time_systemclock",
    "bundle.activator" : true,
    "bundle.name" : "Service Time SystemClock",
    "bundle.description" : "This bundle uses std::chrono::system_clock"
}
```

Because our bundle provides an activator, we also need to state its existence by setting the `bundle.activator` key to `true`. The last two elements are purely informational and not used directly.

## 3.5 Consuming a Service

The process to consume a service is very similar to the process for publishing a service, except that consumers need to handle some additional error cases.

Again, we use a bundle activator to execute code on bundle start that retrieves and consumes a `ServiceTime` service:

```cpp
#include <cppmicroservices/BundleActivator.h>
#include <cppmicroservices/BundleContext.h>
```
```cpp
#include <cppmicroservices/GetBundleContext.h>
#include <ServiceTime.h>
#include <iostream>
using namespace cppmicroservices;

class ServiceTimeConsumerActivator : public BundleActivator
{
  typedef ServiceReference<ServiceTime> ServiceTimeRef;

  void Start(BundleContext ctx)
  {
    auto ref = ctx.GetServiceReference<ServiceTime>();
    PrintTime(ref);
  }

  void Stop(BundleContext)
  {
    // Nothing to do
  }

  void PrintTime(const ServiceTimeRef& ref) const
  {
    if (!ref) {
      std::cout << "ServiceTime reference invalid" << std::endl;
      return;
    }

    // We can also get the bundle context like this
    auto ctx = GetBundleContext();

    // Get the ServiceTime service
    auto svc_time = ctx.GetService(ref);
    if (!svc_time) {
      std::cout << "ServiceTime not available" << std::endl;
    } else {
      std::cout << "Elapsed: " << svc_time->elapsed().count() << "ms" << std::endl;
    }
  }
};
CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(ServiceTimeConsumerActivator)

Because the C++ Micro Services is a dynamic environment, a particular service might not be available yet. Therefore, we first need to check the validity of some returned objects.

The above code would be sufficient only in the simplest use cases. To avoid bundle start ordering problems (e.g. one bundle assuming the existence of a service published by another bundle), a ServiceTracker should be used instead. Such a tracker allows bundles to react on service events and in turn be more robust.

The CMake code for creating a library containing the bundle is very similar to the code for the publishing bundle and thus not included here.

3.5. Consuming a Service
3.6 Installing and Starting Bundles

The two bundles above are embedded in separate libraries and need to be installed into a Framework and started. This is done by a small example program:

```cpp
#include <cppmicroservices/Bundle.h>
#include <cppmicroservices/BundleContext.h>
#include <cppmicroservices/BundleImport.h>
#include <cppmicroservices/Framework.h>
#include <cppmicroservices/FrameworkFactory.h>

using namespace cppmicroservices;

int main(int argc, char* argv[]) {
    #ifdef US_BUILD_SHARED_LIBS
    if (argc < 2) {
        std::cout << "Pass shared libraries as command line arguments" << std::endl;
    }
    #endif

    // Create a new framework with a default configuration.
    Framework fw = FrameworkFactory().NewFramework();

    try {
        // Initialize the framework, such that we can call
        // GetBundleContext() later.
        fw.Init();
        catch (const std::exception& e) {
            std::cout << e.what() << std::endl;
            return 1;
        }

        // The framework inherits from the Bundle class; it is
        // itself a bundle.
        auto ctx = fw.GetBundleContext();
        if (!ctx) {
            std::cerr << "Invalid framework context" << std::endl;
            return 1;
        }

        // Install all bundles contained in the shared libraries
        // given as command line arguments.
        for (int i = 1; i < argc; ++i) {
            try {
                ctx.InstallBundles(argv[i]);
            } catch (const std::exception& e) {
                std::cerr << e.what() << std::endl;
            }
        }

        try {
            // Start the framework itself.
            fw.Start();

            // Our bundles depend on each other in the sense that the consumer
            // bundle expects a ServiceTime service in its activator Start()
            // function. This is done here for simplicity, but is actually
```
// bad practice.
auto bundles = ctx.GetBundles();
auto iter = std::find_if(bundles.begin(), bundles.end(), [](Bundle& b) {
    return b.GetSymbolicName() == "service_time_systemclock";
});
if (iter != bundles.end()) {
    iter->Start();
}

// Now start all bundles.
for (auto& bundle : bundles) {
    bundle.Start();
}

} catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
}

return 0;

#if !defined(US_BUILD_SHARED_LIBS)
CPPMICROSERVICES_IMPORT_BUNDLE(service_time_systemclock)
CPPMICROSERVICES_IMPORT_BUNDLE(service_time_consumer)
#endif

The program expects a list of file system paths pointing to installable libraries. It will first construct a new Framework instance and then install the given libraries. Next, it will start all available bundles.

When the Framework instance is destroyed, it will automatically shut itself down, essentially stopping all active bundles.

See also:
A more detailed tutorial demonstrating some more advanced features is also available.
Contributing to C++ Micro Services

This page contains information about reporting issues as well as some tips and guidelines useful to experienced open source contributors. Make sure you read the contribution guideline before you start participating.

4.1 Reporting issues

A great way to contribute to the project is to send a detailed report when you encounter an issue. We always appreciate a well-written, thorough bug report, and will thank you for it!

Check that the issue database doesn’t already include that problem or suggestion before submitting an issue. If you find a match, add a quick “+1” or “I have this problem too.” Doing this helps prioritize the most common problems and requests.

When reporting issues, please include your host OS and compiler vendor and version. Please also include the steps required to reproduce the problem if possible and applicable.

4.2 CppMicroServices RFCs

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Some changes though are “substantial”, and we ask that these be put through a bit of a design process and produce a consensus among the CppMicroServices core team.

The “RFC” (request for comments) process is intended to provide a consistent and controlled path for new features to enter the framework.

Please refer to the RFC repository for more information.
4.3 Contribution guidelines

This section gives the experienced contributor some tips and guidelines.

4.3.1 Conventions

Fork the repository and make changes on your fork in a feature branch:

- If it’s a bug fix branch, name it XXXX-something where XXXX is the number of the issue.
- If it’s a feature branch, create an enhancement issue to announce your intentions, and name it XXXX-something where XXXX is the number of the issue.

Code must be formatted according to our `.clang-format` file, using the clang-format tool.

Submit unit tests for your changes.

Update the documentation when creating or modifying features. Test your documentation changes for clarity, concision, and correctness.

Pull request descriptions should be as clear as possible and include a reference to all the issues that they address. If the pull request is a result of a RFC process, include the link to the corresponding RFC pull request.

Commit messages must start with a capitalized and short summary (max. 50 chars) written in the imperative, followed by an optional, more detailed explanatory text which is separated from the summary by an empty line.

Code review comments may be added to your pull request. Discuss, then make the suggested modifications and push additional commits to your feature branch. Post a comment after pushing. New commits show up in the pull request automatically, but the reviewers are notified only when you comment.

Pull requests must be cleanly rebased on top of development without multiple branches mixed into the PR.

**Tip:** Git tip: If your PR no longer merges cleanly, use rebase development in your feature branch to update your pull request rather than merge development.

Before you make a pull request, squash your commits into logical units of work using git rebase -i and git push -f. A logical unit of work is a consistent set of patches that should be reviewed together: for example, upgrading the version of a vendored dependency and taking advantage of its now available new feature constitute two separate units of work. Implementing a new function and calling it in another file constitute a single logical unit of work. The very high majority of submissions should have a single commit, so if in doubt: squash down to one.

After every commit, make sure the test suite passes. Include documentation changes in the same pull request so that a revert would remove all traces of the feature or fix.

Include an issue reference like Closes #XXXX or Fixes #XXXX in commits that close an issue. Including references automatically closes the issue on a merge.

If your change is large enough to warrant a copyright statement, add yourself to the COPYRIGHT file, using the same style as the existing entries.

4.3.2 Sign your work

The sign-off is a simple line at the end of the explanation for the patch. Your signature certifies that you wrote the patch or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below (from developercertificate.org):

```plaintext
Tip: Git tip: If your PR no longer merges cleanly, use rebase development in your feature branch to update your pull request rather than merge development.

Before you make a pull request, squash your commits into logical units of work using git rebase -i and git push -f. A logical unit of work is a consistent set of patches that should be reviewed together: for example, upgrading the version of a vendored dependency and taking advantage of its now available new feature constitute two separate units of work. Implementing a new function and calling it in another file constitute a single logical unit of work. The very high majority of submissions should have a single commit, so if in doubt: squash down to one.

After every commit, make sure the test suite passes. Include documentation changes in the same pull request so that a revert would remove all traces of the feature or fix.

Include an issue reference like Closes #XXXX or Fixes #XXXX in commits that close an issue. Including references automatically closes the issue on a merge.

If your change is large enough to warrant a copyright statement, add yourself to the COPYRIGHT file, using the same style as the existing entries.

4.3.2 Sign your work

The sign-off is a simple line at the end of the explanation for the patch. Your signature certifies that you wrote the patch or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below (from developercertificate.org):
```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
660 York Street, Suite 102,
San Francisco, CA 94110 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or

(b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

Then you just add a line to every git commit message:

Signed-off-by: Joe Smith <joe.smith@email.com>

Use your real name (sorry, no pseudonyms or anonymous contributions).

If you set your user.name and user.email git configs, you can sign your commit automatically with git commit -s.

4.4 Contributor Covenant Code of Conduct

4.4.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.
4.4.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

• Using welcoming and inclusive language
• Being respectful of differing viewpoints and experiences
• Gracefully accepting constructive criticism
• Focusing on what is best for the community
• Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

• The use of sexualized language or imagery and unwelcome sexual attention or advances
• Trolling, insulting/derogatory comments, and personal or political attacks
• Public or private harassment
• Publishing others’ private information, such as a physical or electronic address, without explicit permission
• Other conduct which could reasonably be considered inappropriate in a professional setting

4.4.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

4.4.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

4.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at info@cppmicroservices.org. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project’s leadership.

4.4.6 Attribution

5.1 Copyright

Source: https://github.com/CppMicroServices/CppMicroServices

Files: * (see the VCS history for detailed change information)
Copyright: 2015 Sascha Zelzer <sascha.zelzer@gmail.com>
Copyright: 2012 - 2015 Sascha Zelzer, German Cancer Research Center <sascha.zelzer@dkfz-heidelberg.de>
License: Apache License 2.0

Files: * (see the VCS history for detailed change information)
License: Apache License 2.0

Files: core/include/usAny.h
Copyright: Kevlin Henney, 2000, 2001, 2002. All rights reserved.
Extracted from Boost 1.46.1 and adapted for CppMicroServices.
License: See the file header

Files: third_party/*
Copyright: See third_party/README
License: See third_party/README

5.2 License

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the license.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.
You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.
To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.
This tutorial creates successively more complex bundles to illustrate most of the features and functionality offered by the C++ Micro Services library. It is heavily based on the Apache Felix OSGi Tutorial.

### 6.1 Example 1 - Service Event Listener

This example creates a simple bundle that listens for service events. This example does not do much at first, because it only prints out the details of registering and unregistering services. In the next example we will create a bundle that implements a service, which will cause this bundle to actually do something. For now, we will just use this example to help us understand the basics of creating a bundle and its activator.

A bundle gains access to the C++ Micro Services API using a unique instance of `cppmicroservices::BundleContext`. In order for a bundle to get its unique bundle context, it must call `GetBundleContext()` or implement the `cppmicroservices::BundleActivator` interface. This interface has two methods, `Start()` and `Stop()`, that both receive the bundle’s context and are called when the bundle is started and stopped, respectively.

In the following source code, our bundle implements the `BundleActivator` interface and uses the context to add itself as a listener for service events (in the `eventlistener/Activator.cpp` file):

```cpp
#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/Constants.h"
#include "cppmicroservices/ServiceEvent.h"

#include <iostream>

using namespace cppmicroservices;

namespace {

    /**
     * This class implements a simple bundle that utilizes the CppMicroServices's
     * event mechanism to listen for service events. Upon receiving a service event,
     */
```
* it prints out the event’s details.
*/

class Activator : public BundleActivator
{

private:
/**
 * Implements BundleActivator::Start(). Prints a message and adds a member
 * function to the bundle context as a service listener.
 *
 * @param context the framework context for the bundle.
 */

void Start(BundleContext context)
{
    std::cout << "Starting to listen for service events." << std::endl;
    listenerToken = context.AddServiceListener(
        std::bind(&Activator::ServiceChanged, this, std::placeholders::_1));
}

/**
 * Implements BundleActivator::Stop(). Prints a message and removes the
 * member function from the bundle context as a service listener.
 *
 * @param context the framework context for the bundle.
 */

void Stop(BundleContext context)
{
    context.RemoveListener(std::move(listenerToken));
    std::cout << "Stopped listening for service events." << std::endl;

    // Note: It is not required that we remove the listener here,
    // since the framework will do it automatically anyway.
}

/**
 * Prints the details of any service event from the framework.
 *
 * @param event the fired service event.
 */

void ServiceChanged(const ServiceEvent& event)
{
    std::string objectClass =
        ref_any_cast<std::vector<std::string>>(
            event.GetServiceReference().GetProperty(Constants::OBJECTCLASS)).
            front();

    if (event.GetType() == ServiceEvent::SERVICE_REGISTERED) {
        std::cout << "Ex1: Service of type " << objectClass << " registered." << std::endl;
    } else if (event.GetType() == ServiceEvent::SERVICE_UNREGISTERING) {
        std::cout << "Ex1: Service of type " << objectClass << " unregistered." << std::endl;
    } else if (event.GetType() == ServiceEvent::SERVICE_MODIFIED) {
        std::cout << "Ex1: Service of type " << objectClass << " modified." << std::endl;
    }
}
After implementing the C++ source code for the bundle activator, we must export the activator as shown in the last line above. This ensures that the C++ Micro Services library can create an instance of the activator and call the `Start()` and `Stop()` methods.

After implementing the source code for the bundle, we must also define a manifest file that contains meta-data needed by the C++ Micro Services framework for manipulating the bundle. The manifest is embedded in the shared library along with the compiled source code. We create a file called `manifest.json` that contains the following:

```json
{
  "bundle.symbolic_name" : "eventlistener",
  "bundle.activator" : true
}
```

Next, we need to compile the source code. This example uses CMake as the build system and the top-level CMake-Lists.txt file could look like this:

```cmake
# [prj-start]
project(CppMicroServicesExamples)
set(CMAKE_CXX_STANDARD_REQUIRED 1)
set(CMAKE_CXX_STANDARD 17)
find_package(CppMicroServices NO_MODULE REQUIRED)
cmake_minimum_required(VERSION $(US_CMAKE_MINIMUM_REQUIRED_VERSION))
cmake_policy(VERSION $(US_CMAKE_MINIMUM_REQUIRED_VERSION))
# [prj-end]

#---------------------------------------------------------------
# Init output directories
#---------------------------------------------------------------
set(CppMicroServicesExamples_ARCHIVE_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}/lib")
set(CppMicroServicesExamples_LIBRARY_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}/lib")
set(CppMicroServicesExamples_RUNTIME_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}/bin")
foreach(_type ARCHIVE LIBRARY RUNTIME)
  if(NOT CMAKE_${_type}_OUTPUT_DIRECTORY)
    set(CMAKE_${_type}_OUTPUT_DIRECTORY ${CppMicroServicesExamples_${_type}_OUTPUT_DIRECTORY})
  endif()
endforeach()

function(CreateTutorial _name)
  set(_srcs ${ARGN})
  usFunctionGetResourceSource(TARGET Tutorial-$_name OUT _srcs)
  usFunctionGenerateBundleInit(TARGET Tutorial-$_name OUT _srcs)
  add_library(Tutorial-$_name ${_srcs})

#---------------------------------------------------------------
```
set_property(TARGET Tutorial-$_name) APPEND PROPERTY COMPILE_DEFINITIONS US_BUNDLE_NAME=${_name})
set_property(TARGET Tutorial-$_name) PROPERTY DEBUG_POSTFIX "")

if($_nameDEPENDS)
  foreach(_dep $_nameDEPENDS)
    include_directories($_PROJECT_SOURCE_DIR/tutorial/$(_dep))
    target_link_libraries(Tutorial-$(_name) Tutorial-$(_dep))
  endforeach()
endif()
target_link_libraries(Tutorial-$(_name) ${CppMicroServices_LIBRARIES})
set_target_properties(Tutorial-$(_name) PROPERTIES
  LABELS Tutorial
  OUTPUT_NAME $(_name)
)
usFunctionAddResources(TARGET Tutorial-$_name) BUNDLE_NAME $(_name) WORKING_DIRECTORY $(_PROJECT_SOURCE_DIR)/tutorial/$(_name)/resources FILES manifest.json
usFunctionEmbedResources(TARGET Tutorial-$_name)
endfunction()
add_subdirectory(eventlistener)

and the CMakeLists.txt file in the eventlistener subdirectory is:

set(_srcs Activator.cpp)
CreateTutorial(eventlistener $_srcs)

The call to usFunctionGenerateBundleInit creates required callback functions to be able to manage the bundle within the C++ Micro Services runtime. If you are not using CMake, you have to place a macro call to CPPMICROSERVICES_INITIALIZE_BUNDLE yourself into the bundle’s source code, e.g. in Activator.cpp. Have a look at Build Instructions for more details about using CMake or other build systems (e.g. Makefiles) when writing bundles.

To run the examples contained in the C++ Micro Services library, we use a small driver program called usTutorialDriver:

CppMicroServices-build> bin/usTutorialDriver
> h
This help text
start <id | name> Start the bundle with id <id> or name <name>
stop <id | name> Stop the bundle with id <id> or name <name>
status Print status information
shutdown Shut down the framework

Typing status at the command prompt lists all installed bundles and their current state. Note that the driver program pre-installs the example bundles, so they will be listed initially with the INSTALLED state. To start the eventlistener bundle, type start eventlistener at the command prompt:

> status
Id | Symbolic Name | State
-----------------------------------
 0 | system_bundle | ACTIVE
 1 | eventlistener | INSTALLED
 2 | dictionaryservice | INSTALLED
 3 | frenchdictionary | INSTALLED
The above command started the eventlistener bundle (implicitly loading its shared library). Keep in mind, that this bundle will not do much at this point since it only listens for service events and we are not registering any services. In the next example we will register a service that will generate an event for this bundle to receive. To exit the usTutorialDriver, use the shutdown command.

### 6.2 Example 2 - Dictionary Service Bundle

This example creates a bundle that implements a service. Implementing a service is a two-step process, first we must define the interface of the service and then we must define an implementation of the service interface. In this particular example, we will create a dictionary service that we can use to check if a word exists, which indicates if the word is spelled correctly or not. First, we will start by defining a simple dictionary service interface in a file called dictionaryservice/IDictionaryService.h:

```cpp
#include "cppmicroservices/ServiceInterface.h"
#include <string>

#ifdef US_BUILD_SHARED_LIBS
# ifdef Tutorial_dictionaryservice_EXPORTS
# define DICTIONARYSERVICE_EXPORT US_ABI_EXPORT
# else
# define DICTIONARYSERVICE_EXPORT US_ABI_IMPORT
# endif
#else
# define DICTIONARYSERVICE_EXPORT US_ABI_EXPORT
#endif

/**
 * A simple service interface that defines a dictionary service.
 * A dictionary service simply verifies the existence of a word.
 **/
struct DICTIONARYSERVICE_EXPORT IDictionaryService
{
    // Out-of-line virtual destructor for proper dynamic cast
    // support with older versions of gcc.
    virtual ~IDictionaryService();

    /**
     * Check for the existence of a word.
     * @param word the word to be checked.
     * @return true if the word is in the dictionary,
     * false otherwise.
     **/
    virtual bool CheckWord(const std::string& word) = 0;
};
```

6.2. Example 2 - Dictionary Service Bundle 33
The service interface is quite simple, with only one method that needs to be implemented. Because we provide an empty out-of-line destructor (defined in the file IDictionaryService.cpp) we must export the service interface by using the bundle specific DICTIONARYSERVICE_EXPORT macro.

In the following source code, the bundle uses its bundle context to register the dictionary service. We implement the dictionary service as an inner class of the bundle activator class, but we could have also put it in a separate file. The source code for our bundle is as follows in a file called dictionaryservice/Activator.cpp:

```cpp
#include "IDictionaryService.h"
#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/ServiceProperties.h"

#include <memory>
#include <set>

using namespace cppmicroservices;

namespace {

/**
 * This class implements a bundle activator that uses the bundle
 * context to register an English language dictionary service
 * with the C++ Micro Services registry during static initialization
 * of the bundle. The dictionary service interface is
 * defined in a separate file and is implemented by a nested class.
 */

class US_ABI_LOCAL Activator : public BundleActivator {

private:

/**
 * A private inner class that implements a dictionary service;
 * see IDictionaryService for details of the service.
 */

class DictionaryImpl : public IDictionaryService {

   // The set of words contained in the dictionary.
   std::set<std::string> m_dictionary;

public:

   DictionaryImpl() {
      m_dictionary.insert("welcome");
      m_dictionary.insert("to");
      m_dictionary.insert("the");
      m_dictionary.insert("micro");
      m_dictionary.insert("services");
      m_dictionary.insert("tutorial");
   }

   /**
    * Implements IDictionaryService::CheckWord(). Determines
    * if the passed in word is contained in the dictionary.
    * @param word the word to be checked.
    * @return true if the word is in the dictionary,
    * false otherwise.
   */

```
```cpp
bool CheckWord(const std::string& word)
{
    std::string lword(word);
    std::transform(lword.begin(), lword.end(), lword.begin(), ::tolower);

    return m_dictionary.find(lword) != m_dictionary.end();
}

public:

/**
 * Implements BundleActivator::Start(). Registers an
 * instance of a dictionary service using the bundle context;
 * attaches properties to the service that can be queried
 * when performing a service look-up.
 * @param context the context for the bundle.
 */
void Start(BundleContext context)
{
    std::shared_ptr<DictionaryImpl> dictionaryService =
        std::make_shared<DictionaryImpl>();
    ServiceProperties props;
    props["Language"] = std::string("English");
    context.RegisterService<IDictionaryService>(dictionaryService, props);
}

/**
 * Implements BundleActivator::Stop(). Does nothing since
 * the C++ Micro Services library will automatically unregister any registered
 * services.
 * @param context the context for the bundle.
 */
void Stop(BundleContext /*context*/)
{
    // NOTE: The service is automatically unregistered
}

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(Activator)
```

Note that we do not need to unregister the service in the Stop() method, because the C++ Micro Services library will automatically do so for us. The dictionary service that we have implemented is very simple; its dictionary is a set of only five words, so this solution is not optimal and is only intended for educational purposes.

**Note:** In this example, the service interface and implementation are both contained in one bundle which exports the interface class. However, service implementations almost never need to be exported and in many use cases it is beneficial to provide the service interface and its implementation(s) in separate bundles. In such a scenario, clients of a service will only have a link-time dependency on the shared library providing the service interface (because of the out-of-line destructor) but not on any bundles containing service implementations.

We must create a manifest.json file that contains the meta-data for our bundle; the manifest file contains the following:

```json
{
    "bundle.symbolic_name" : "dictionaryservice",
```
"bundle.activator" : true

For an introduction how to compile our source code, see Example 1 - Service Event Listener.

After running the usTutorialDriver program we should make sure that the bundle from Example 1 is active. We can use the status shell command to get a list of all bundles, their state, and their bundle identifier number. If the Example 1 bundle is not active, we should start the bundle using the start command and the bundle’s identifier number or symbolic name that is displayed by the status command. Now we can start our dictionary service bundle by typing the start dictionaryservice command:

```
CppMicroServices-build> bin/usTutorialDriver
> status
Id | Symbolic Name | State
-----------------------------------
0 | system_bundle | ACTIVE
1 | eventlistener | INSTALLED
2 | dictionaryservice | INSTALLED
3 | frenchdictionary | INSTALLED
4 | dictionaryclient | INSTALLED
5 | dictionaryclient2 | INSTALLED
6 | dictionaryclient3 | INSTALLED
7 | spellcheckservice | INSTALLED
8 | spellcheckclient | INSTALLED
> start eventlistener
Starting to listen for service events.
> start dictionaryservice
Ex1: Service of type IDictionaryService registered.
> status
Id | Symbolic Name | State
-----------------------------------
0 | system_bundle | ACTIVE
1 | eventlistener | ACTIVE
2 | dictionaryservice | ACTIVE
3 | frenchdictionary | INSTALLED
4 | dictionaryclient | INSTALLED
5 | dictionaryclient2 | INSTALLED
6 | dictionaryclient3 | INSTALLED
7 | spellcheckservice | INSTALLED
8 | spellcheckclient | INSTALLED
>
```

To stop the bundle, use the stop 2 command. If the bundle from Example 1 is still active, then we should see it print out the details of the service event it receives when our new bundle registers its dictionary service. Using the usTutorialDriver commands stop and start we can stop and start it at will, respectively. Each time we start and stop our dictionary service bundle, we should see the details of the associated service event printed from the bundle from Example 1. In Example 3, we will create a client for our dictionary service. To exit usTutorialDriver, we use the shutdown command.

### 6.3 Example 2b - Alternative Dictionary Service Bundle

This example creates an alternative implementation of the dictionary service defined in Example 2. The source code for the bundle is identical except that:

- Instead of using English words, it uses French words.
• We do not need to define the dictionary service interface again, as we can just link the definition from the bundle in Example 2.

The main point of this example is to illustrate that multiple implementations of the same service may exist; this example will also be of use to us in Example 5.

In the following source code, the bundle uses its bundle context to register the dictionary service. We implement the dictionary service as an inner class of the bundle activator class, but we could have also put it in a separate file. The source code for our bundle is as follows in a file called dictionaryclient/Activator.cpp:

```cpp
#include "IDictionaryService.h"
#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/ServiceProperties.h"

#include <algorithm>
#include <memory>
#include <set>
using namespace cppmicroservices;
namespace {

/**
 * This class implements a bundle activator that uses the bundle context to register a French language dictionary service with the C++ Micro Services registry during static initialization of the bundle. The dictionary service interface is defined in Example 2 (dictionaryservice) and is implemented by a nested class. This class is identical to the class in Example 2, except that the dictionary contains French words.
 */
class US_ABI_LOCAL Activator : public BundleActivator
{

private:

 /**
 * A private inner class that implements a dictionary service; see DictionaryService for details of the service.
 */
class DictionaryImpl : public IDictionaryService
{
  // The set of words contained in the dictionary.
  std::set<std::string> m_dictionary;

public:
  DictionaryImpl()
  {
    m_dictionary.insert("bienvenue");
    m_dictionary.insert("au");
    m_dictionary.insert("tutoriel");
    m_dictionary.insert("micro");
    m_dictionary.insert("services");
  }

  /**
   * Implements DictionaryService.checkWord(). Determines if the passed in word is contained in the dictionary.
   */
```
We must create a `manifest.json` file that contains the meta-data for our bundle; the manifest file contains the following:

```json
{
    "bundle.symbolic_name" : "frenchdictionary",
    "bundle.activator" : true
}
```

For a refresher on how to compile our source code, see *Example 1 - Service Event Listener*. Because we use the `IDictionaryService` definition from Example 2, we also need to make sure that the proper include paths and linker dependencies are set:
set(_srcs Activator.cpp)
set(frenchdictionary_DEPENDS dictionaryservice)
CreateTutorial(frenchdictionary ${_srcs})

After running the usTutorialDriver program, we should make sure that the bundle from Example 1 is active. We can use the status shell command to get a list of all bundles, their state, and their bundle identifier number. If the Example 1 bundle is not active, we should start the bundle using the start command and the bundle’s identifier number or name that is displayed by the status command. Now we can start our dictionary service bundle by typing the start frenchdictionary command:

    CppMicroServices-build> bin/usTutorialDriver
    > status
    Id   | Symbolic Name     | State
    -----------------------------
    0    | system_bundle      | ACTIVE
    1    | eventlistener      | INSTALLED
    2    | dictionaryservice  | INSTALLED
    3    | frenchdictionary   | INSTALLED
    4    | dictionaryclient   | INSTALLED
    5    | dictionaryclient2  | INSTALLED
    6    | dictionaryclient3  | INSTALLED
    7    | spellcheckservice  | INSTALLED
    8    | spellcheckclient   | INSTALLED
    > start eventlistener
    Starting to listen for service events.
    > start frenchdictionary
    Ex1: Service of type IDictionaryService registered.
    > status
    Id   | Symbolic Name     | State
    -----------------------------
    0    | system_bundle      | ACTIVE
    1    | eventlistener      | ACTIVE
    2    | dictionaryservice  | INSTALLED
    3    | frenchdictionary   | ACTIVE
    4    | dictionaryclient   | ACTIVE
    5    | dictionaryclient2  | INSTALLED
    6    | dictionaryclient3  | INSTALLED
    7    | spellcheckservice  | INSTALLED
    8    | spellcheckclient   | INSTALLED
    >

To stop the bundle, use the stop 3 command. If the bundle from Example 1 is still active, then we should see it print out the details of the service event it receives when our new bundle registers its dictionary service. Using the usTutorialDriver commands stop and start we can stop and start it at will, respectively. Each time we start and stop our dictionary service bundle, we should see the details of the associated service event printed from the bundle from Example 1. In Example 3, we will create a client for our dictionary service. To exit usTutorialDriver, we use the shutdown command.

Note: Because our French dictionary bundle has a link dependency on the dictionary service bundle from Example 2, this bundle’s shared library is automatically loaded by the operating system’s dynamic loader. However, its status remains INSTALLED until it is explicitly started.
6.4 Example 3 - Dictionary Client Bundle

This example creates a bundle that is a client of the dictionary service implemented in Example 2. In the following source code, our bundle uses its bundle context to query for a dictionary service. Our client bundle uses the first dictionary service it finds, and if none are found, it prints a message and stops. Services operate with no additional overhead. The source code for our bundle is as follows in a file called dictionaryclient/Activator.cpp:

```cpp
#include "IDictionaryService.h"
#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include <iostream>

using namespace cppmicroservices;

namespace {

/**
 * This class implements a bundle activator that uses a dictionary service to check
 * for
 * the proper spelling of a word by check for its existence in the dictionary.
 * This bundles uses the first service that it finds and does not monitor the
 * dynamic availability of the service (i.e., it does not listen for the arrival
 * or departure of dictionary services). When starting this bundle, the thread
 * calling the Start() method is used to read words from standard input. You can
 * stop checking words by entering an empty line, but to start checking words
 * again you must unload and then load the bundle again.
 */

class US_ABI_LOCAL Activator : public BundleActivator {

public:

/**
 * Implements BundleActivator::Start(). Queries for all available dictionary
 * services. If none are found it simply prints a message and returns,
 * otherwise it reads words from standard input and checks for their
 * existence from the first dictionary that it finds.
 * 
 * @param context the bundle context for this bundle.
 */

void Start(BundleContext context) {

// Query for all service references matching any language.
std::vector<ServiceReference<IDictionaryService>> refs =
    context.GetServiceReferences<IDictionaryService>("(Language=*)");

if (!refs.empty()) {
    std::cout << "Enter a blank line to exit." << std::endl;

    // Loop endlessly until the user enters a blank line
    while (std::cin) {
        // Ask the user to enter a word.
        std::cout << "Enter word: ";
    }
}
}

//...
std::string word;
std::getline(std::cin, word);

// If the user entered a blank line, then
// exit the loop.
if (word.empty()) {
    break;
}

// First, get a dictionary service and then check
// if the word is correct.
std::shared_ptr<IDictionaryService> dictionary =
    context.GetService<IDictionaryService>(refs.front());
if (dictionary->CheckWord(word)) {
    std::cout << "Correct." << std::endl;
} else {
    std::cout << "Incorrect." << std::endl;
}
} else {
    std::cout << "Couldn't find any dictionary service..." << std::endl;
}
}

/**
 * Implements BundleActivator::Stop(). Does nothing since
 * the C++ Micro Services library will automatically unget any used services.
 * @param context the context for the bundle.
 */
void Stop(BundleContext /*context*/) {
    // NOTE: The service is automatically released.
};

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(Activator)

Note that we do not need to unget or release the service in the Stop() method, because the C++ Micro Services library will automatically do so for us. We must create a manifest.json file with the meta-data for our bundle, which contains the following:

{  
    "bundle.symbolic_name" : "dictionaryclient",
    "bundle.activator" : true
}

Since we are using the IDictionaryService interface defined in Example 1, we must link our bundle to the dictionaryservice bundle:

set(_srcs Activator.cpp)
set(dictionaryclient_DEPENDS dictionaryservice)
CreateTutorial(dictionaryclient ${_srcs})

After running the usTutorialDriver executable, and starting the event listener bundle, we can use the start dictionaryclient command to start our dictionary client bundle:
The above command starts the pre-installed bundle. When we start the bundle, it will use the main thread to prompt us for words. Enter one word at a time to check the words, and enter a blank line to stop checking words. To reload the bundle, we must first use the `stop dictionaryclient` command to stop the bundle, then the `start dictionaryclient` command to re-start it. To test the dictionary service, enter any of the words in the dictionary (e.g., “welcome”, “to”, “the”, “micro”, “services”, “tutorial”) or any word not in the dictionary.

This example client is simple enough and, in fact, is too simple. What would happen if the dictionary service were to unregister suddenly? Our client would abort with a segmentation fault due to a null pointer access when trying to use the service object. This dynamic service availability issue is a central tenant of the service model. As a result, we must make our client more robust in dealing with such situations. In Example 4, we explore a slightly more complicated dictionary client that dynamically monitors service availability.

## 6.5 Example 4 - Robust Dictionary Client Bundle

In Example 3, we create a simple client bundle for our dictionary service. The problem with that client was that it did not monitor the dynamic availability of the dictionary service, thus an error would occur if the dictionary service disappeared while the client was using it. In this example we create a client for the dictionary service that monitors the dynamic availability of the dictionary service. The result is a more robust client.

The functionality of the new dictionary client is essentially the same as the old client, it reads words from standard input and checks for their existence in the dictionary service. Our bundle uses its bundle context to register itself as a service event listener; monitoring service events allows the bundle to monitor the dynamic availability of the dictionary service. Our client uses the first dictionary service it finds. The source code for our bundle is as follows in a file called `Activator.cpp`:

```cpp
#include "IDictionaryService.h"
#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/Constants.h"
#include "cppmicroservices/ServiceEvent.h"
#include <iostream>

using namespace cppmicroservices;

namespace {

/**
 * This class implements a bundle activator that uses a dictionary service to check for
 * the proper spelling of a word by checking for its existence in the dictionary. This bundle is more complex than the bundle in Example 3 because
 * it monitors the dynamic availability of the dictionary services. In other words, if the service it is using departs, then it stops using it gracefully,
 * or if it needs a service and one arrives, then it starts using it automatically. As before, the bundle uses the first service that it finds and
*/
```
class US_ABI_LOCAL Activator : public BundleActivator
{

public:
    Activator()
    : m_context(),
      m_dictionary(nullptr)
    {}  

    /**
     * Implements BundleActivator::Start(). Adds itself as a listener for service 
     * events, then queries for available dictionary services. If any 
     * dictionaries are found it gets a reference to the first one available and 
     * then starts its "word checking loop". If no dictionaries are found, then 
     * it just goes directly into its "word checking loop", but it will not be 
     * able to check any words until a dictionary service arrives; any arriving 
     * dictionary service will be automatically used by the client if a 
     * dictionary is not already in use. Once it has dictionary, it reads words 
     * from standard input and checks for their existence in the dictionary that 
     * it is using. 
     * 
     * \note It is very bad practice to use the calling thread to perform a 
     * lengthy process like this; this is only done for the purpose of 
     * the tutorial. 
     * 
     * @param context the bundle context for this bundle. 
     */
    void Start(BundleContext context)
    {
        m_context = context;
        
        // Use your favorite thread library to synchronize member 
        // variable access within this scope while registering 
        // the service listener and performing our initial 
        // dictionary service lookup since we 
        // don't want to receive service events when looking up the 
        // dictionary service, if one exists. 
        // MutexLocker lock(&m_mutex); 

        // Listen for events pertaining to dictionary services.
        m_context.AddServiceListener(
            std::bind(&Activator::ServiceChanged, this, std::placeholders::_1),
            std::string("{&(") + Constants::OBJECTCLASS + ")" + 
                us_service_interface_iid<IDictionaryService>() + ")") + 
                "(Language=*)")
            );

        // Query for any service references matching any language.
        std::vector<ServiceReference<IDictionaryService>> refs = context.GetServiceReferences<IDictionaryService>("(Language=*)");

        // If we found any dictionary services, then just get 
        // a reference to the first one so we can use it.
        if (!refs.empty()) 
            
            // Use your favorite thread library to synchronize member 
            // variable access within this scope while registering 
            // the service listener and performing our initial 
            // dictionary service lookup since we 
            // don't want to receive service events when looking up the 
            // dictionary service, if one exists. 
            // MutexLock mutex(&m_mutex);

            // Listen for events pertaining to dictionary services.
            m_context.AddServiceListener(
                std::bind(&Activator::ServiceChanged, this, std::placeholders::_1),
                std::string("{&(") + Constants::OBJECTCLASS + ")" + 
                    us_service_interface_iid<IDictionaryService>() + ")") + 
                    "(Language=*)")
                );

            // Query for any service references matching any language.
            std::vector<ServiceReference<IDictionaryService>> refs = context.GetServiceReferences<IDictionaryService>("(Language=*)");

            // If we found any dictionary services, then just get 
            // a reference to the first one so we can use it.
            if (!refs.empty()) 

            // Use your favorite thread library to synchronize member 
            // variable access within this scope while registering 
            // the service listener and performing our initial 
            // dictionary service lookup since we 
            // don't want to receive service events when looking up the 
            // dictionary service, if one exists. 
            // MutexLock mutex(&m_mutex);

            // Listen for events pertaining to dictionary services.
            m_context.AddServiceListener(
                std::bind(&Activator::ServiceChanged, this, std::placeholders::_1),
                std::string("{&(") + Constants::OBJECTCLASS + ")" + 
                    us_service_interface_iid<IDictionaryService>() + ")") + 
                    "(Language=*)")
                );

            // Query for any service references matching any language.
            std::vector<ServiceReference<IDictionaryService>> refs = context.GetServiceReferences<IDictionaryService>("(Language=*)");

            // If we found any dictionary services, then just get 
            // a reference to the first one so we can use it.
            if (!refs.empty()) 

            // Use your favorite thread library to synchronize member 
            // variable access within this scope while registering 
            // the service listener and performing our initial 
            // dictionary service lookup since we 
            // don't want to receive service events when looking up the 
            // dictionary service, if one exists. 
            // MutexLock mutex(&m_mutex);

            // Listen for events pertaining to dictionary services.
            m_context.AddServiceListener(
                std::bind(&Activator::ServiceChanged, this, std::placeholders::_1),
                std::string("{&(") + Constants::OBJECTCLASS + ")" + 
                    us_service_interface_iid<IDictionaryService>() + ")") + 
                    "(Language=*)")
                );

            // Query for any service references matching any language.
            std::vector<ServiceReference<IDictionaryService>> refs = context.GetServiceReferences<IDictionaryService>("(Language=*)");

            // If we found any dictionary services, then just get 
            // a reference to the first one so we can use it.
m_ref = refs.front();
m_dictionary = m_context.GetService(m_ref);
}
}

std::cout << "Enter a blank line to exit." << std::endl;

// Loop endlessly until the user enters a blank line
while (std::cin) {
    // Ask the user to enter a word.
    std::cout << "Enter word: ";

    std::string word;
    std::getline(std::cin, word);

    // If the user entered a blank line, then
    // exit the loop.
    if (word.empty()) {
        break;
    }

    // If there is no dictionary, then say so.
    else if (m_dictionary == nullptr) {
        std::cout << "No dictionary available." << std::endl;
    }
    // Otherwise print whether the word is correct or not.
    else if (m_dictionary->CheckWord(word)) {
        std::cout << "Correct." << std::endl;
    } else {
        std::cout << "Incorrect." << std::endl;
    }
}

/**
 * Implements BundleActivator::Stop(). Does nothing since
 * the C++ Micro Services library will automatically unget any used services.
 * @param context the context for the bundle.
 */
void Stop(BundleContext /*context*/) {
    // NOTE: The service is automatically released.
}

/**
 * Implements ServiceListener.serviceChanged(). Checks to see if the service
 * we are using is leaving or tries to get a service if we need one.
 * @param event the fired service event.
 */
void ServiceChanged(const ServiceEvent& event) {
    // Use your favorite thread library to synchronize this
    // method with the Start() method.
    MutexLocker lock(&m_mutex);

    // If a dictionary service was registered, see if we
    // need one. If so, get a reference to it.
    if (event.GetType() == ServiceEvent::SERVICE_REGISTERED) {
    }
if (!m_ref) {
    // Get a reference to the service object.
    m_ref = event.GetServiceReference();
    m_dictionary = m_context.GetService(m_ref);
}

// If a dictionary service was unregistered, see if it was the one we were using. If so, unget the service and try to query to get another one.
else if (event.GetType() == ServiceEvent::SERVICE_UNREGISTERING) {
    if (event.GetServiceReference() == m_ref) {
        // Unget service object and null references.
        m_ref = nullptr;
        m_dictionary.reset();

        // Query to see if we can get another service.
        std::vector<ServiceReference<IDictionaryService>> refs;
        try {
            refs = m_context.GetServiceReferences<IDictionaryService>("(Language=*)");
        }
        catch (const std::invalid_argument& e) {
            std::cout << e.what() << std::endl;
        }
        if (!refs.empty()) {
            // Get a reference to the first service object.
            m_ref = refs.front();
            m_dictionary = m_context.GetService(m_ref);
        }
    }
}

private:
// Bundle context
BundleContext m_context;

// The service reference being used
ServiceReference<IDictionaryService> m_ref;

// The service object being used
std::shared_ptr<IDictionaryService> m_dictionary;
};

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(Activator)

The client listens for service events indicating the arrival or departure of dictionary services. If a new dictionary service arrives, the bundle will start using that service if and only if it currently does not have a dictionary service. If an existing dictionary service disappears, the bundle will check to see if the disappearing service is the one it is using; if it is it stops using it and tries to query for another dictionary service, otherwise it ignores the event.

Like normal, we must create a `manifest.json` file that contains the meta-data for our bundle:

```json
{
    "bundle.symbolic_name" : "dictionaryclient2",
    "bundle.activator" : true
}
```

6.5. Example 4 - Robust Dictionary Client Bundle
As in Example 3, we must link our bundle to the dictionaryservice bundle:

```
set(_srcs Activator.cpp)
set(dictionaryclient2_DEPENDS dictionaryservice)
CreateTutorial(dictionaryclient2 ${_srcs})
```

After running the usTutorialDriver executable, and starting the event listener bundle, we can use the `start dictionaryclient2` command to start our robust dictionary client bundle:

```
CppMicroServices-debug> bin/usTutorialDriver
> start eventlistener
Starting to listen for service events.
> start dictionaryclient2
Ex1: Service of type IDictionaryService registered.
Enter a blank line to exit.
Enter word:
```

The above command starts the bundle and it will use the main thread to prompt us for words. Enter one word at a time to check the words and enter a blank line to stop checking words. To reload the bundle, we must first use the `stop dictionaryclient2` command to stop the bundle, then the `start dictionaryclient2` command to re-start it. To test the dictionary service, enter any of the words in the dictionary (e.g., “welcome”, “to”, “the”, “micro”, “services”, “tutorial”) or any word not in the dictionary.

Since this client monitors the dynamic availability of the dictionary service, it is robust in the face of sudden departures of the dictionary service. Further, when a dictionary service arrives, it automatically gets the service if it needs it and continues to function. These capabilities are a little difficult to demonstrate since we are using a simple single-threaded approach, but in a multi-threaded or GUI-oriented application this robustness is very useful.

### 6.6 Example 5 - Service Tracker Dictionary Client Bundle

In Example 4, we created a more robust client bundle for our dictionary service. Due to the complexity of dealing with dynamic service availability, even that client may not sufficiently address all situations. To deal with this complexity the C++ Micro Services library provides the `cppmicroservices::ServiceTracker` utility class. In this example we create a client for the dictionary service that uses the `ServiceTracker` class to monitor the dynamic availability of the dictionary service, resulting in an even more robust client.

The functionality of the new dictionary client is essentially the same as the one from Example 4. Our bundle uses its bundle context to create a `ServiceTracker` instance to track the dynamic availability of the dictionary service on our behalf. Our client uses the dictionary service returned by the `ServiceTracker`, which is selected based on a ranking algorithm defined by the C++ Micro Services library. The source code for our bundles is as follows in a file called `dictionaryclient3/Activator.cpp`:

```cpp
#include "IDictionaryService.h"
#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/ServiceTracker.h"

using namespace cppmicroservices;

namespace {

    /**
     * This class implements a bundle activator that uses a dictionary
     * service to check for the proper spelling of a word by
     */
```
* checking for its existence in the dictionary. This bundle
* uses a service tracker to dynamically monitor the availability
* of a dictionary service, instead of providing a custom service
* listener as in Example 4. The bundle uses the service returned
* by the service tracker, which is selected based on a ranking
* algorithm defined by the C++ Micro Services library.
* Again, the calling thread of the Start() method is used to read
* words from standard input, checking its existence in the dictionary.
* You can stop checking words by entering an empty line, but
* to start checking words again you must unload and then load
* the bundle again.
*/

class US_ABI_LOCAL Activator : public BundleActivator
{

public:

Activator()
    : m_context(), m_tracker(nullptr)
{

/**
 * Implements BundleActivator::Start(). Creates a service
 * tracker to monitor dictionary services and starts its "word
 * checking loop". It will not be able to check any words until
 * the service tracker finds a dictionary service; any discovered
 * dictionary service will be automatically used by the client.
 * It reads words from standard input and checks for their
 * existence in the discovered dictionary.
 *
 * \note It is very bad practice to use the calling thread to perform a
 * lengthy process like this; this is only done for the purpose of
 * the tutorial.
 *
 * @param context the bundle context for this bundle.
 */

void Start(BundleContext context)
{

m_context = context;

// Create a service tracker to monitor dictionary services.

m_tracker = new ServiceTracker<IDictionaryService>(
    m_context,
    LDAPFilter(std::string("(&("
       + Constants::OBJECTCLASS + "+" +
       us_service_interface_iid<IDictionaryService>() + ")" +
       "(Language=*))"));

m_tracker->Open();

std::cout << "Enter a blank line to exit." << std::endl;

// Loop endlessly until the user enters a blank line
while (std::cin) {
    // Ask the user to enter a word.
    std::cout << "Enter word: ";

    std::string word;
    std::getline(std::cin, word);

    // Check the word in the dictionary.
    // ...
// Get the selected dictionary, if available.
std::shared_ptr<IDictionaryService> dictionary = m_tracker->GetService();

// If the user entered a blank line, then
// exit the loop.
if (word.empty()) {
    break;
}

// If there is no dictionary, then say so.
else if (!dictionary) {
    std::cout << "No dictionary available." << std::endl;
}

// Otherwise print whether the word is correct or not.
else if (dictionary->CheckWord(word)) {
    std::cout << "Correct." << std::endl;
} else {
    std::cout << "Incorrect." << std::endl;
}

// This automatically closes the tracker
delete m_tracker;

/**
 * Implements BundleActivator::Stop(). Does nothing since
 * the C++ Micro Services library will automatically unget any used services.
 * @param context the context for the bundle.
 */
void Stop(BundleContext /*context*/) {}

private:
// Bundle context
BundleContext m_context;

// The service tracker
ServiceTracker<IDictionaryService>* m_tracker;
};

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(Activator)

Since this client uses the ServiceTracker utility class, it will automatically monitor the dynamic availability of the dictionary service. Like normal, we must create a manifest.json file that contains the meta-data for our bundle:

```json
{
    "bundle.symbolic_name" : "dictionaryclient3",
    "bundle.activator" : true
}
```

Again, we must link our bundle to the dictionaryservice bundle:

```bash
set(_srcs Activator.cpp)
set(dictionaryclient3_DEPENDS dictionaryservice)
CreateTutorial(dictionaryclient3 ${_srcs})
```

After running the usTutorialDriver executable, and starting the event listener bundle, we can use the start
dictionaryclient3 command to start our robust dictionary client bundle:

```plaintext
CppMicroServices-debug> bin/usTutorialDriver
> start eventlistener
> start dictionaryclient3
Ex1: Service of type IDictionaryService registered.
Enter a blank line to exit.
Enter word:
```

The above command starts the bundle and it will use the main thread to prompt us for words. Enter one word at a time to check the words and enter a blank line to stop checking words. To re-start the bundle, we must first use the stop dictionaryclient3 command to stop the bundle, then the start dictionaryclient3 command to re-start it. To test the dictionary service, enter any of the words in the dictionary (e.g., “welcome”, “to”, “the”, “micro”, “services”, “tutorial”) or any word not in the dictionary.

Since this client monitors the dynamic availability of the dictionary service, it is robust in the face of sudden departures of the dictionary service. Further, when a dictionary service arrives, it automatically gets the service if it needs it and continues to function. These capabilities are a little difficult to demonstrate since we are using a simple single-threaded approach, but in a multi-threaded or GUI-oriented application this robustness is very useful.

### 6.7 Example 6 - Spell Checker Service Bundle

In this example, we complicate things further by defining a new service that uses an arbitrary number of dictionary services to perform its function. More precisely, we define a spell checker service which will aggregate all dictionary services and provide another service that allows us to spell check passages using our underlying dictionary services to verify the spelling of words. Our bundle will only provide the spell checker service if there are at least two dictionary services available. First, we will start by defining the spell checker service interface in a file called spellcheckservice/ISpellCheckService.h:

```c++
#include "cppmicroservices/ServiceInterface.h"
#include <string>
#include <vector>

#ifdef US_BUILD_SHARED_LIBS
# ifdef Tutorial_spellcheckservice_EXPORTS
# define SPELLCHECKSERVICE_EXPORT US_ABI_EXPORT
# else
# define SPELLCHECKSERVICE_EXPORT US_ABI_IMPORT
# endif
#else
# define SPELLCHECKSERVICE_EXPORT US_ABI_EXPORT
#endif

/**
 * A simple service interface that defines a spell check service. A spell check
 * service checks the spelling of all words in a given passage. A passage is any
 * number of words separated by a space character and the following punctuation
 * marks: comma, period, exclamation mark, question mark, semi-colon, and colon.
 */
struct SPELLCHECKSERVICE_EXPORT ISpellCheckService
{
    // Out-of-line virtual destructor for proper dynamic cast
    // support with older versions of gcc.
    virtual ~ISpellCheckService();
```
The service interface is quite simple, with only one method that needs to be implemented. Because we provide an empty out-of-line destructor (defined in the file ISpellCheckService.cpp) we must export the service interface by using the bundle specific SPELLCHECKSERVICE_EXPORT macro.

In the following source code, the bundle needs to create a complete list of all dictionary services; this is somewhat tricky and must be done carefully if done manually via service event listeners. Our bundle makes use of the cppmicroservices::ServiceTracker and cppmicroservices::ServiceTrackerCustomizer classes to robustly react to service events related to dictionary services. The bundle activator of our bundle now additionally implements the ServiceTrackerCustomizer class to be automatically notified of arriving, departing, or modified dictionary services. In case of a newly added dictionary service, our ServiceTrackerCustomizer::AddingService() implementation checks if a spell checker service was already registered and if not registers a new ISpellCheckService instance if at least two dictionary services are available. If the number of dictionary services drops below two, our ServiceTrackerCustomizer implementation un-registers the previously registered spell checker service instance. These actions must be performed in a synchronized manner to avoid interference from service events originating from different threads. The implementation of our bundle activator is done in a file called spellcheckservice/Activator.cpp:

```cpp
#include "IDICTIONARYSERVICE.h"
#include "ISPELLCHECKSERVICE.h"

#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/ServiceTracker.h"
#include "cppmicroservices/ServiceTrackerCustomizer.h"

#include <cstring>
#include <map>
#include <memory>

US_MSVC_PUSH_DISABLE_WARNING(4996)

using namespace cppmicroservices;

namespace {

    /**
     * This class implements a bundle that implements a spell
     * checker service. The spell checker service uses all available
     * dictionary services to check for the existence of words in
     * a given sentence. This bundle uses a ServiceTracker to
     * monitors the dynamic availability of dictionary services,
     * and to aggregate all available dictionary services as they
     * arrive and depart. The spell checker service is only registered
     * if there are dictionary services available, thus the spell
```
checker service will appear and disappear as dictionary services appear and disappear, respectively.

```cpp
class US_ABI_LOCAL Activator : public BundleActivator, public ServiceTrackerCustomizer<IDictionaryService>
{

private:
    /**<
     * A private inner class that implements a spell check service; see
     * ISpellCheckService for details of the service.
     */
    class SpellCheckImpl : public ISpellCheckService
    {

private:
    typedef std::map<ServiceReference<IDictionaryService>,
        std::shared_ptr<IDictionaryService>>
        RefToServiceType;
    RefToServiceType m_refToSvcMap;

public:
    /**<
     * Implements ISpellCheckService::Check(). Checks the given passage for
     * misspelled words.
     * @param passage the passage to spell check.
     * @return A list of misspelled words.
     */
    std::vector<std::string> Check(const std::string& passage)
    {
        std::vector<std::string> errorList;

        // No misspelled words for an empty string.
        if (passage.empty()) {
            return errorList;
        }

        // Tokenize the passage using spaces and punctuation.
        const char* delimiters = ".,!?;:");
        char* passageCopy = new char[passage.size() + 1];
        std::memcpy(passageCopy, passage.c_str(), passage.size() + 1);
        char* pch = std::strtok(passageCopy, delimiters);

        // Lock the m_refToSvcMap member using your favorite thread library here...
        // MutexLocker lock(&m_refToSvcMapMutex)

        // Loop through each word in the passage.
        while (pch) {
            std::string word(pch);

            bool correct = false;

            // Check each available dictionary for the current word.
            for (RefToServiceType::const_iterator i = m_refToSvcMap.begin();
                (!correct) && (i != m_refToSvcMap.end()));
```
++i) {
    std::shared_ptr<IDictionaryService> dictionary = i->second;

    if (dictionary->CheckWord(word)) {
        correct = true;
    }

    // If the word is not correct, then add it
    // to the incorrect word list.
    if (!correct) {
        errorList.push_back(word);
    }

    pch = std::strtok(nullptr, delimiters);
}
}

delete[] passageCopy;

return errorList;
}

std::size_t AddDictionary(const ServiceReference<IDictionaryService>& ref, std::shared_ptr<IDictionaryService> dictionary) {
    // Lock the m_refToSvcMap member using your favorite thread library here...
    // MutexLocker lock(&m_refToSvcMapMutex)
    m_refToSvcMap.insert(std::make_pair(ref, dictionary));

    return m_refToSvcMap.size();
}

std::size_t RemoveDictionary(const ServiceReference<IDictionaryService>& ref) {
    // Lock the m_refToSvcMap member using your favorite thread library here...
    // MutexLocker lock(&m_refToSvcMapMutex)
    m_refToSvcMap.erase(ref);

    return m_refToSvcMap.size();
};

virtual std::shared_ptr<IDictionaryService> AddingService(const ServiceReference<IDictionaryService>& reference) {
    std::shared_ptr<IDictionaryService> dictionary = m_context.GetService(reference);
    std::size_t count = m_spellCheckService->AddDictionary(reference, dictionary);
    if (!m_spellCheckReg && count > 1) {
        m_spellCheckReg = m_context.RegisterService<ISpellCheckService>(m_spellCheckService);
    }

    return dictionary;
virtual void ModifiedService(
    const ServiceReference<IDictionaryService>& /*reference*/,
    const std::shared_ptr<IDictionaryService>& /*service*/)
{
    // do nothing
}

virtual void RemovedService(
    const ServiceReference<IDictionaryService>& reference,
    const std::shared_ptr<IDictionaryService>& /*service*/)
{
    if (m_spellCheckService->RemoveDictionary(reference) < 2 &&
        m_spellCheckReg) {
        m_spellCheckReg.Unregister();
        m_spellCheckReg = nullptr;
    }
}

std::shared_ptr<SpellCheckImpl> m_spellCheckService;
ServiceRegistration<ISpellCheckService> m_spellCheckReg;
BundleContext m_context;
std::unique_ptr<ServiceTracker<IDictionaryService>> m_tracker;

public:
Activator()
    : m_context()
{}

/**
 * Implements BundleActivator::Start(). Registers an
 * instance of a dictionary service using the bundle context;
 * attaches properties to the service that can be queried
 * when performing a service look-up.
 * @param context the context for the bundle.
 */
void Start(BundleContext context)
{
    m_context = context;

    m_spellCheckService.reset(new SpellCheckImpl);
    m_tracker.reset(new ServiceTracker<IDictionaryService>(context, this));
    m_tracker->Open();
}

/**
 * Implements BundleActivator::Stop(). Does nothing since
 * the C++ Micro Services library will automatically unregister any registered
 * services and release any used services.
 * @param context the context for the bundle.
 */
void Stop(BundleContext /*context*/)
{
Note that we do not need to unregister the service in `Stop()` method, because the C++ Micro Services library will automatically do so for us. The spell checker service that we have implemented is very simple; it simply parses a given passage into words and then loops through all available dictionary services for each word until it determines that the word is correct. Any incorrect words are added to an error list that will be returned to the caller. This solution is not optimal and is only intended for educational purposes. Next, we create a `manifest.json` file that contains the meta-data for our bundle:

```json
{
    "bundle.symbolic_name": "spellcheckservice",
    "bundle.activator": true
}
```

Note: In this example, the service interface and implementation are both contained in one bundle which exports the interface class. However, service implementations almost never need to be exported and in many use cases it is beneficial to provide the service interface and its implementation(s) in separate bundles. In such a scenario, clients of a service will only have a link-time dependency on the shared library providing the service interface (because of the out-of-line destructor) but not on any bundles containing service implementations. This often leads to bundles which do not export any symbols at all.

For an introduction how to compile our source code, see Example 1 - Service Event Listener.

After running the `usTutorialDriver` program we should make sure that the bundle from Example 1 is active. We can use the `status` shell command to get a list of all bundles, their state, and their bundle identifier number. If the Example 1 bundle is not active, we should start the bundle using the start command and the bundle’s identifier number or symbolic name that is displayed by the `status` command. Now we can start the spell checker service bundle by entering the `start spellcheckservice` command which will also trigger the starting of the dictionaryservice bundle containing the english dictionary:

```
CppMicroServices-build> bin/usTutorialDriver
> start eventlistener
Starting to listen for service events.
> start spellcheckservice
> status
Id | Symbolic Name         | State               
-------------------------------------------
 0 | system_bundle          | ACTIVE              
 1 | eventlistener          | ACTIVE              
 2 | dictionaryservice      | INSTALLED           
 3 | frenchdictionary       | INSTALLED           
 4 | dictionaryclient       | INSTALLED           
 5 | dictionaryclient2      | INSTALLED           
 6 | dictionaryclient3      | INSTALLED           
 7 | spellcheckservice      | ACTIVE              
 8 | spellcheckclient       | INSTALLED           
```
To trigger the registration of the spell checker service from our bundle, we start the frenchdictionary using the `start frenchdictionary` command. If the bundle from Example 1 is still active, then we should see it print out the details of the service event it receives when our new bundle registers its spell checker service:

```bash
CppMicroServices-build> bin/usTutorialDriver
> start frenchdictionary
Ex1: Service of type IDictionaryService registered.
Ex1: Service of type ISpellCheckService registered.
>
```

We can experiment with our spell checker service’s dynamic availability by stopping the french dictionary service; when the service is stopped, the eventlistener bundle will print that our bundle is no longer offering its spell checker service. Likewise, when the french dictionary service comes back, so will our spell checker service. We create a client for our spell checker service in Example 7. To exit the `usTutorialDriver` program, we use the `shutdown` command.

### 6.8 Example 7 - Spell Checker Client Bundle

In this example we create a client for the spell checker service we implemented in Example 6. This client monitors the dynamic availability of the spell checker service using the Service Tracker and is very similar in structure to the dictionary client we implemented in Example 5. The functionality of the spell checker client reads passages from standard input and spell checks them using the spell checker service. Our bundle uses its bundle context to create a `ServiceTracker` object to monitor spell checker services. The source code for our bundle is as follows in a file called `spellcheckclient/Activator.cpp`:

```cpp
#include "ISpellCheckService.h"
#include "cppmicroservices/BundleActivator.h"
#include "cppmicroservices/BundleContext.h"
#include "cppmicroservices/ServiceTracker.h"

#include <cstring>
#include <iostream>

using namespace cppmicroservices;

namespace {

/**
 * This class implements a bundle that uses a spell checker service to check the spelling of a passage. This bundle is essentially identical to Example 5, in that it uses the Service Tracker to monitor the dynamic availability of the spell checker service. When starting this bundle, the thread calling the Start() method is used to read passages from standard input. You can stop spell checking passages by entering an empty line, but to start spell checking again you must un-load and then load the bundle again.
 ***/

class US_ABI_LOCAL Activator : public BundleActivator {

public:
```
Activator()
: m_context()
, m_tracker(nullptr)
()

/**
 * Implements BundleActivator::Start(). Creates a service
 * tracker object to monitor spell checker services. Enters
 * a spell check loop where it reads passages from standard
 * input and checks their spelling using the spell checker service.
 * 
 * \note It is very bad practice to use the calling thread to perform a
 * lengthy process like this; this is only done for the purpose of
 * the tutorial.
 * 
 * @param context the bundle context for this bundle.
 */
void Start(BundleContext context)
{
   m_context = context;

   // Create a service tracker to monitor spell check services.
   m_tracker = new ServiceTracker<ISpellCheckService>(m_context);
   m_tracker->Open();

   //std::cout << "Tracker count is :" << m_tracker->GetTrackingCount() << std::endl;
   std::cout << "Enter a blank line to exit." << std::endl;

   // Loop endlessly until the user enters a blank line
   while (std::cin) {
      // Ask the user to enter a passage.
      std::cout << "Enter passage: ";

      std::string passage;
      std::getline(std::cin, passage);

      // Get the selected spell check service, if available.
      std::shared_ptr<ISpellCheckService> checker = m_tracker->GetService();

      // If the user entered a blank line, then
      // exit the loop.
      if (passage.empty()) {
         break;
      }
      // If there is no spell checker, then say so.
      else if (checker == nullptr) {
         std::cout << "No spell checker available." << std::endl;
      }
      // Otherwise check passage and print misspelled words.
      else {
         std::vector<std::string> errors = checker->Check(passage);

         if (errors.empty()) {
            std::cout << "Passage is correct." << std::endl;
         } else {
            std::cout << "Incorrect word(s):" << std::endl;
            for (std::size_t i = 0; i < errors.size(); ++i) {
               std::cout << " " << errors[i] << std::endl;
            }
         }
      }
   }
}
// This automatically closes the tracker
delete m_tracker;
}

/**
 * Implements BundleActivator::Stop(). Does nothing since
 * the C++ Micro Services library will automatically unget any used services.
 * @param context the context for the bundle.
 */
void Stop(BundleContext /*context*/) {}

private:
// Bundle context
BundleContext m_context;

// The service tracker
ServiceTracker<ISpellCheckService>* m_tracker;
};

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(Activator)

After running the usTutorialDriver program use the status command to make sure that only the bundles from Example 2, Example 2b, and Example 6 are started; use the start (start <id | name>) and stop (stop <id | name>) commands as appropriate to start and stop the various tutorial bundles, respectively. Now we can start our spell checker client bundle by entering start spellcheckclient:

```
CppMicroServices-build> bin/usTutorialDriver
> start eventlistener
Starting to listen for service events.
> start spellcheckservice
> status
Id | Symbolic Name | State
-------------------------------
 0 | system_bundle   | ACTIVE
 1 | eventlistener   | ACTIVE
 2 | dictionaryservice | INSTALLED
 3 | frenchdictionary | INSTALLED
 4 | dictionaryclient | INSTALLED
 5 | dictionaryclient2 | INSTALLED
 6 | dictionaryclient3 | INSTALLED
 7 | spellcheckservice | ACTIVE
 8 | spellcheckclient | INSTALLED
>
```

To trigger the registration of the spell checker service from our bundle, we start the frenchdictionary using the start frenchdictionary command. If the bundle from Example 1 is still active, then we should see it print out the details of the service event it receives when our new bundle registers its spell checker service:

```
CppMicroServices-build> bin/usTutorialDriver
> start spellcheckservice
> start frenchdictionary
```

6.8. Example 7 - Spell Checker Client Bundle
> start spellcheckclient
Enter a blank line to exit.
Enter passage:

When we start the bundle, it will use the main thread to prompt us for passages; a passage is a collection of words separated by spaces, commas, periods, exclamation points, question marks, colons, or semi-colons. Enter a passage and press the enter key to spell check the passage or enter a blank line to stop spell checking passages. To restart the bundle, we must first use the `stop` command to stop the bundle, then the `start` command to re-start it.

Since this client uses the Service Tracker to monitor the dynamic availability of the spell checker service, it is robust in the scenario where the spell checker service suddenly departs. Further, when a spell checker service arrives, it automatically gets the service if it needs it and continues to function. These capabilities are a little difficult to demonstrate since we are using a simple single-threaded approach, but in a multi-threaded or GUI-oriented application this robustness is very useful.
Integrating C++ Micro Services into an existing code-base can be done incrementally, e.g. by starting to convert class singletons to services.

### 7.1 Meyers Singleton

Singletons are a well known pattern to ensure that only one instance of a class exists during the whole life-time of the application. A self-deleting variant is the “Meyers Singleton”:

```cpp
class SingletonOne
{
public:
    static SingletonOne& GetInstance();

    // Just some member
    int a;

private:
    SingletonOne();
    ~SingletonOne();

    // Disable copy constructor and assignment operator.
    SingletonOne(const SingletonOne&);
    SingletonOne& operator=(const SingletonOne&);
};
```

where the `getInstance()` method is implemented as

```cpp
SingletonOne& SingletonOne::getInstance()
{
    static SingletonOne instance;
    return instance;
}
```
If such a singleton is accessed during static deinitialization, your program might crash or even worse, exhibit undefined behavior, depending on your compiler and/or weekday. Such an access might happen in destructors of other objects with static life-time.

For example, suppose that SingletonOne needs to call a second Meyers singleton during destruction:

```cpp
SingletonOne::~SingletonOne()
{
    std::cout << "SingletonTwo::b = " << SingletonTwo::GetInstance().b << std::endl;
}
```

If SingletonTwo was destroyed before SingletonOne, this leads to the mentioned problems. Note that this problem only occurs for static objects defined in the same shared library.

Since you cannot reliably control the destruction order of global static objects, you must not introduce dependencies between them during static deinitialization. This is one reason why one should consider an alternative approach to singletons (unless you can absolutely make sure that nothing in your shared library will introduce such dependencies. Never.)

Of course you could use something like a Phoenix singleton but that will have other drawbacks in certain scenarios. Returning pointers instead of references in GetInstance() would open up the possibility to return a nullptr, but than again this would not help if you require a non-NULL instance in your destructor.

Another reason for an alternative approach is that singletons are usually not meant to be singletons for eternity. If your design evolves, you might hit a point where you suddenly need multiple instances of your singleton.

### 7.2 Singletons as a Service

C++ Micro Services can be used to emulate the singleton pattern using a non-singleton class. This leaves room for future extensions without the need for heavy refactoring. Additionally, it gives you full control about the construction and destruction order of your “singletons” inside your shared library or executable, making it possible to have dependencies between them during destruction.

#### 7.2.1 Converting a Classic Singleton

We modify the previous SingletonOne class such that it internally uses the micro services API. The changes are discussed in detail below.

```cpp
class SingletonOneService
{
    public:
        // This will return a SingletonOneService instance with the
        // lowest service id at the time this method was called the first
        // time and returned a non-null value (which is usually the instance
        // which was registered first). An empty object is returned if no
        // instance was registered yet.
        //
        // Note: This is a helper method to migrate traditional singletons to
        // services. Do not create a method like this in real world applications.
        static std::shared_ptr<SingletonOneService> GetInstance();

        int a;

    SingletonOneService();

}```
~SingletonOneService();

private:
    // Disable copy constructor and assignment operator.
    SingletonOneService(const SingletonOneService&);
    SingletonOneServices operator=(const SingletonOneService&);
};

• In the implementation above, the class SingletonOneService provides the implementation as well as the interface.

• The SingletonOneService class looks like a plain C++ class, no need for hiding constructors and destructor.

Let's have a look at the modified GetInstance() and ~SingletonOneService() methods.

```
std::shared_ptr<SingletonOneService> SingletonOneService::GetInstance()
{
    static ServiceReference<SingletonOneService> serviceRef;
    static auto context = GetBundleContext();

    if (!serviceRef) {
        // This is either the first time GetInstance() was called,
        // or a SingletonOneService instance has not yet been registered.
        serviceRef = context.GetServiceReference<SingletonOneService>();
    }

    if (serviceRef) {
        // We have a valid service reference. It always points to the service
        // with the lowest id (usually the one which was registered first).
        // This still might return a null pointer, if all SingletonOneService
        // instances have been unregistered (during unloading of the library,
        // for example).
        return context.GetService(serviceRef);
    } else {
        // No SingletonOneService instance was registered yet.
        return nullptr;
    }
}
```

The inline comments should explain the details. Note that we now had to change the return type to a shared pointer, instead of a reference as in the classic singleton. This is necessary since we can no longer guarantee that an instance always exists. Clients of the GetInstance() method must check if the returned object is empty and react appropriately.

**Note:** Newly created “singletons” should not expose a GetInstance() method. They should be handled as proper services and hence should be retrieved by clients using the BundleContext or ServiceTracker API. The GetInstance() method is for migration purposes only.

```
SingletonOneService::~SingletonOneService()
{
    std::shared_ptr<SingletonTwoService> singletonTwoService =
        SingletonTwoService::GetInstance();

    // The bundle activator must ensure that a SingletonTwoService instance is
    // available during destruction of a SingletonOneService instance.
```
assert(singletonTwoService != nullptr);
std::cout << "SingletonTwoService::b = " << singletonTwoService->b
<< std::endl;
}

The SingletonTwoService::GetInstance() method is implemented exactly as in SingletonOneService. Because we know that the bundle activator guarantees that a SingletonTwoService instance will always be available during the life-time of a SingletonOneService instance (see below), we can assert a non-null pointer. Otherwise, we would have to handle the null-pointer case.

The order of construction/registration and destruction/unregistration of our singletons (or any other services) is defined in the Start() and Stop() methods of the bundle activator.

```cpp
void Start(BundleContext context)
{
    // First create and register a SingletonTwoService instance.
    m_SingletonTwo = std::make_shared<SingletonTwoService>();
    m_SingletonTwoReg =
        context.RegisterService<SingletonTwoService>(m_SingletonTwo);
    // Framework service registry has shared ownership of the SingletonTwoService
    // instance

    // Now the SingletonOneService constructor will get a valid SingletonTwoService instance.
    m_SingletonOne = std::make_shared<SingletonOneService>();
    m_SingletonOneReg =
        context.RegisterService<SingletonOneService>(m_SingletonOne);
}
```

The Stop() method is defined as:

```cpp
void Stop(BundleContext /*context*/) {
    // Services are automatically unregistered during unloading of the shared library after the call to Stop(BundleContext*) has returned.

    // Since SingletonOneService needs a non-null SingletonTwoService instance in its destructor, we explicitly unregister and delete the SingletonOneService instance here. This way, the SingletonOneService destructor will still get a valid SingletonTwoService instance.
    m_SingletonOneReg.Unregister();
    m_SingletonOne.reset();
    // Deletion of the SingletonTwoService instance is handled by the smart pointer

    // For singletonTwoService, we could rely on the automatic unregistering by the service registry and on automatic deletion of service instances through smart pointers.
    m_SingletonTwoReg.Unregister();
    m_SingletonTwo.reset();
    // Deletion of the SingletonOneService instance is handled by the smart pointer
}
```
8.1 The Resource System

The C++ Micro Services library provides a generic resource system that allows you to:

- Embed resources in a bundle.
- Access resources at runtime.

The features and limitations of the resource system are described in more detail in the following sections.

8.1.1 Embedding Resources in a Bundle

Resources are embedded into a bundle’s shared or static library (or into an executable) by using the `usResourceCompiler3` executable. It will create a ZIP archive of all input files and can append it to the bundle file with a configurable compression level. See `usResourceCompiler3` for the command line reference.

8.1.2 Accessing Resources at Runtime

Each bundle provides individual resource lookup and access to its embedded resources via the `Bundle` class which provides methods returning `BundleResource` objects. The `BundleResource` class provides a high-level API for accessing resource information and traversing the resource tree.

The `BundleResourceStream` class provides a `std::istream` compatible object for the seamless usage of embedded resource data in third-party libraries.

Resources are managed in a tree hierarchy, modeling the original parent-child relationship on the file-system.

The following example shows how to retrieve a resource from each currently installed bundle whose path is specified by a bundle property:

```c++
// Check if a bundle defines a "service-component" property
// and use its value to retrieve an embedded resource containing
// a component description.
```
```cpp
for (auto const& bundle : bundleCtx.GetBundles()) {
    if (bundle.GetState() == Bundle::STATE_UNINSTALLED)
        continue;
    auto headers = bundle.GetHeaders();
    auto iter = headers.find("service-component");
    std::string componentPath =
        (iter == headers.end()) ? std::string() : iter->second.ToString();
    if (!componentPath.empty()) {
        BundleResource componentResource = bundle.GetResource(componentPath);
        if (!componentResource.IsValid() || componentResource.IsDir())
            continue;

        // Create a std::istream compatible object and parse the
        // component description.
        BundleResourceStream resStream(componentResource);
        parseComponentDefinition(resStream);
    }
}
```

This example could be enhanced to dynamically react to bundles being started and stopped, making use of the popular extender pattern from OSGi.

### 8.1.3 Runtime Overhead

The resource system has the following runtime characteristics:

- During bundle install, the bundle’s ZIP archive header data (if available) is parsed and stored in memory.
- Querying Bundle or BundleResource objects for resource information will not extract the embedded resource data and hence only has minimal runtime and memory overhead.
- Creating a BundleResourceStream object will allocate memory for the uncompressed resource data and inflate it. The memory will be free’ed after the BundleResourceStream object is destroyed.

### 8.1.4 Conventions and Limitations

- Resources have a size limitation of 2GB due to the use of the ZIP format.
- Resource entries are stored with case-insensitive names. On case-sensitive file systems, adding resources with the same name but different capitalization will lead to an error.
- Looking up resources by name at runtime is case sensitive.
- The CppMicroServices library will search for a valid zip file inside a shared library, starting from the end of the file. If other zip files are embedded in the bundle as well (e.g. as an additional resource embedded via the Windows RC compiler or using other techniques), it will stop at the first valid zip file and use it as the resource container.

### 8.2 Bundle Properties

A C++ Micro Services Bundle provides meta-data in the form of so-called properties about itself. Properties are key-value pairs where the key is of type std::string and the value of type Any. The following properties are always set by the C++ Micro Services library and cannot be altered by the bundle author:

- bundle.id - The unique id of the bundle (type long)
• **bundle.location** - The full path to the bundle’s shared library on the file system (type `std::string`)

Bundle authors must always add the following property to their bundle’s `manifest.json` file:

• **bundle.symbolic_name** - The human readable name of the bundle (type `std::string`)

C++ Micro Services will not install any bundle which doesn’t contain a valid ‘bundle.symbolic_name’ property in its `manifest.json` file.

Bundle authors can add custom properties by providing a `manifest.json` file, embedded as a top-level resource into the bundle (see *The Resource System*). The root value of the JSON file must be a JSON object. An example `manifest.json` file would be:

```json
{
    "bundle.symbolic_name": "my bundle",
    "bundle.version": "1.0.2",
    "bundle.description": "This bundle provides an awesome service",
    "authors": [ "John Doe", "Douglas Reynolds", "Daniel Cannady" ],
    "rating": 5
}
```

All JSON member names of the root object will be available as property keys in the bundle containing the `manifest.json` file. The C++ Micro Services library specifies the following standard keys for re-use in `manifest.json` files:

• **bundle.version** - The version of the bundle (type `std::string`). The version string must be a valid version identifier, as specified in the BundleVersion class.

• **bundle.vendor** - The vendor name of the bundle (type `std::string`)

• **bundle.description** - A description for the bundle (type `std::string`)

**Note:** Some of the properties mentioned above may also be accessed via dedicated methods in the Bundle class, e.g. `GetSymbolicName()` or `GetVersion()`.

**Attention:** Despite JSON being a case-sensitive format, C++ Micro Services stores bundle properties as case-insensitive to accommodate LDAP queries using `LDAPFilter` and `LDAPProp`. Either keep the JSON case-insensitive or standardize on a convention to ensure queries return the correct results.

When parsing the `manifest.json` file, the JSON types are mapped to C++ types and stored in instances of the `Any` class. The mapping is as follows:

<table>
<thead>
<tr>
<th>JSON</th>
<th>C++ (Any)</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td><code>std::map</code></td>
</tr>
<tr>
<td>array</td>
<td><code>std::vector</code></td>
</tr>
<tr>
<td>string</td>
<td><code>std::string</code></td>
</tr>
<tr>
<td>number</td>
<td><code>int</code> or <code>double</code></td>
</tr>
<tr>
<td>true</td>
<td><code>bool</code></td>
</tr>
<tr>
<td>false</td>
<td><code>bool</code></td>
</tr>
<tr>
<td>null</td>
<td><code>Any()</code></td>
</tr>
</tbody>
</table>

### 8.3 Service Hooks

The CppMicroServices library implements the Service Hook Service Specification Version 1.1 from OSGi Core Release 5 for C++. Below is a summary of the concept - consult the OSGi specifications for more details.
Service hooks provide mechanisms for bundle writers to closely interact with the CppMicroServices service registry. These mechanisms are not intended for use by application bundles but rather by bundles in need of hooking into the service registry and modifying the behaviour of application bundles.

Some example use cases for service hooks include:

- Proxying of existing services by hiding the original service and registering a proxy service with the same properties
- Providing services on demand based on registered service listeners from external bundles

### 8.3.1 Event Listener Hook

A bundle can intercept events being delivered to other bundles by registering a `ServiceEventListenerHook` object as a service. The CppMicroServices library will send all service events to all the registered hooks using the reversed ordering of their ServiceReference objects.

Note that event listener hooks are called after the event was created, but before it is filtered by the optional filter expression of the service listeners. Therefore, an event listener hook receives all `SERVICE_REGISTERED`, `SERVICE_MODIFIED`, `SERVICE_UNREGISTERING`, and `SERVICE_MODIFIED_ENDMATCH` events regardless of the presence of a service listener filter. It may then remove bundles or specific service listeners from the `ServiceEventListenerHook::ShrinkableMapType` object passed to the `ServiceEventListenerHook::Event()` method to hide service events.

Implementers of the Event Listener Hook must ensure that bundles continue to see a consistent set of service events.

### 8.3.2 Find Hook

Find Hook objects registered using the `ServiceFindHook` interface will be called when bundles look up service references via the `BundleContext::GetServiceReference()` or `BundleContext::GetServiceReferences()` methods. The order in which the CppMicroServices library calls the find hooks is the reverse operator< ordering of their ServiceReference objects. The hooks may remove service references from the `ShrinkableVector` object passed to the `ServiceFindHook::Find()` method to hide services from specific bundles.

### 8.3.3 Listener Hook

The CppMicroServices API provides information about the registration, unregistration, and modification of services. However, it does not directly allow the introspection of bundles to get information about what services a bundle is waiting for.

Bundles may need to wait for a service to arrive (via a registered service listener) before performing their functions. Listener Hooks provide a mechanism to get informed about all existing, newly registered, and removed service listeners.

A Listener Hook object registered using the `ServiceListenerHook` interface will be notified about service listeners by being passed `ServiceListenerHook::ListenerInfo` objects. Each `ListenerInfo` object is related to the registration / unregistration cycle of a specific service listener. That is, registering the same service listener again (even with a different filter) will automatically unregister the previous registration and the newly registered service listener is related to a different `ListenerInfo` object. `ListenerInfo` objects can be stored in unordered containers and compared with each other- for example, to match `ServiceListenerHook::Added()` and `ServiceListenerHook::Removed()` calls.
The Listener Hooks are called synchronously in the same order of their registration. However, in rare cases the removal of a service listener may be reported before its corresponding addition. To handle this case, the `ListenerInfo::IsRemoved()` method is provided which can be used in the `ServiceListenerHook::Added()` method to detect a delivery that is out of order. A simple strategy is to ignore removed events without corresponding added events and ignore added events where the `ListenerInfo` object is already removed:

```cpp
class MyServiceListenerHook : public ServiceListenerHook
{
private:
    class Tracked
    {
        // Do some work during construction and destruction
    };

    std::unordered_map<ListenerInfo, Tracked> tracked;

public:
    void Added(const std::vector<ListenerInfo>& listeners)
    {
        for (std::vector<ListenerInfo>::const_iterator iter = listeners.begin(), endIter = listeners.end(); iter != endIter; ++iter) {
            if (iter->IsRemoved())
                return;
            tracked.insert(std::make_pair(*iter, Tracked()));
        }
    }

    void Removed(const std::vector<ListenerInfo>& listeners)
    {
        for (std::vector<ListenerInfo>::const_iterator iter = listeners.begin(), endIter = listeners.end(); iter != endIter; ++iter) {
            tracked.erase(*iter);
            // If we got a corresponding "Added" event before, the Tracked destructor will do some cleanup...
        }
    }
};
```

### 8.3.4 Architectural Notes

**Ordinary Services**

All service hooks are treated as ordinary services. If the CppMicroServices library uses them, their Service References will show that the CppMicroServices bundles are using them, and if a hook is a Service Factory, then the actual instance will be properly created.

The only speciality of the service hooks is that the CppMicroServices library does not use them for the hooks them-
selves. That is, the Service Event and Service Find Hooks cannot be used to hide the services from the CppMicroSer-
vices library.

**Ordering**

The hooks are very sensitive to ordering because they interact directly with the service registry. In gen-
eral, implementers of the hooks must be aware that other bundles can be started before or after the bundle
which provides the hooks. To ensure early registration of the hooks, they should be registered within the
`BundleActivator::Start()` method of the program executable.

**Multi Threading**

All hooks must be thread-safe because the hooks can be called at any time. All hook methods must be re-entrant, as
they can be entered at any time and in rare cases in the wrong order. The CppMicroServices library calls all hook
methods synchronously, but the calls might be triggered from any user thread interacting with the CppMicroServices
API. The CppMicroServices API can be called from any of the hook methods, but implementers must be careful to
not hold any lock while calling CppMicroServices methods.

### 8.4 Static Bundles

The normal and most flexible way to add a CppMicroServices bundle to an application is to compile it into a shared
library using the `BundleContext::InstallBundles()` function at runtime.

However, bundles can be linked statically to your application or shared library. This makes the deployment of your
application less error-prone and in the case of a complete static build, also minimizes its binary size and start-up time.
However, in order to add new functionality to your application, you must rebuild and redistribute it.

#### 8.4.1 Creating Static Bundles

Static bundles are written just like shared bundles - there are no differences in the usage of the CppMicroServices API
or the provided preprocessor macros.

#### 8.4.2 Using Static Bundles

Static bundles can be used (imported) in shared or other static libraries, or in the executable itself. For every
static bundle you would like to import, you need to add a call to `CPPMICROSERVICES_IMPORT_BUNDLE` or
to `CPPMICROSERVICES_INITIALIZE_STATIC_BUNDLE` (if the bundle does not provide an activator) in the
source code of the importing library.

**Note:** While you can link static bundles to other static bundles, you will still need to import all of the static bundles
into the final executable to ensure proper initialization.

The two main usage scenarios- using a shared or static CppMicroServices library- are explained in the sections below.

**Using a Shared CppMicroServices Library**

Building the CppMicroServices library as a shared library allows you to import static bundles into other shared or
static bundles, or into the executable.
Listing 8.1: Example code for importing `MyStaticBundle1` into another library or executable

```cpp
#include "cppmicroservices/BundleImport.h"
CPPMICROSERVICES_IMPORT_BUNDLE(MyStaticBundle1)
```

Using a Static CppMicroServices Library

The CppMicroServices library can be built as a static library. In that case, creating shared bundles is not supported. If you create shared bundles that link a static version of the CppMicroServices library, the runtime behavior is undefined. In this usage scenario, every bundle will be statically built and linked to an executable:

Listing 8.2: Static bundles and CppMicroServices library

```cpp
#include "cppmicroservices/BundleImport.h"

#ifndef US_BUILD_SHARED_LIBS
CPPMICROSERVICES_INITIALIZE_STATIC_BUNDLE(system_bundle)
CPPMICROSERVICES_IMPORT_BUNDLE(MyStaticBundle2)
CPPMICROSERVICES_INITIALIZE_STATIC_BUNDLE(main)
#endif
```

Note that the first `CPPMICROSERVICES_IMPORT_BUNDLE` call imports the static CppMicroServices library. Next, the `MyStaticBundle2` bundle is imported and finally, the executable itself is initialized (this is necessary if the executable itself is a C++ Micro Services bundle).
CHAPTER 9

Http Service

This bundle is based on the Http Service Specification as detailed in the OSGi Compendium. It allows bundle developers to create communication and user interface solutions based on HTTP and web technologies.

For this purpose, the Http Service bundle provides a servlet architecture similar to the Java Servlet API.

**Warning:** This bundle has not reached a stable version yet. Its design and API may change between releases in a backwards incompatible way.
This bundle provides a HTML based interface to inspect and manage a C++ Micro Services application using a web browser.

It requires the Http Service bundle to publish a servlet that forwards Http requests to the appropriate Web Console plug-in.

**Tip:** After installing and starting the Web Console bundle, go to http://localhost:8080/us/console to access the console.

**Warning:** This bundle has not reached a stable version yet. Its design and API may change between releases in a backwards incompatible way.

### 10.1 Screenshots
Bundles

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Version</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>System Bundle (system_bundle)</td>
<td>3.0.0</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>1</td>
<td>C++ Micro Services Web Console (usWebConsole)</td>
<td>0.1.0</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>2</td>
<td>C++ Micro Services Http Service (usHttpService)</td>
<td>0.1.0</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>3</td>
<td>C++ Micro Services Shell Service (usShellService)</td>
<td>0.1.0</td>
<td>ACTIVE</td>
</tr>
</tbody>
</table>

Fig. 10.1: A list of all installed bundles.

C++ Micro Services Web Console (usWebConsole)

Back to Bundles

<table>
<thead>
<tr>
<th>Service ID</th>
<th>Types</th>
<th>Ranking</th>
<th>Scope</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>[cppmicroservices::HttpServlet]</td>
<td>singleton</td>
<td>▶️ { 4 items }</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>[cppmicroservices::HttpServlet]</td>
<td>singleton</td>
<td>▶️ { 5 items }</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>[cppmicroservices::HttpServlet]</td>
<td>singleton</td>
<td>▶️ { 5 items }</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>[cppmicroservices::HttpServlet]</td>
<td>singleton</td>
<td>▶️ { 5 items }</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10.2: Detailed bundle view.
C++ Micro Services Web Console (usWebConsole)

Back to Bundles

Fig. 10.3: Browse and download bundle resources.

Interface cppmicroservices::HttpServletRequest

Back to Services

<table>
<thead>
<tr>
<th>Bundle</th>
<th>Service ID</th>
<th>Types</th>
<th>Ranking</th>
<th>Scope</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>usWebConsole</td>
<td>5</td>
<td>[cppmicroservices::HttpServletRequest]</td>
<td>singleton</td>
<td>▶️ { 5 items }</td>
<td></td>
</tr>
<tr>
<td>usWebConsole</td>
<td>4</td>
<td>[cppmicroservices::HttpServletRequest]</td>
<td>singleton</td>
<td>▶️ { 5 items }</td>
<td></td>
</tr>
<tr>
<td>usWebConsole</td>
<td>3</td>
<td>[cppmicroservices::HttpServletRequest]</td>
<td>singleton</td>
<td>▶️ { 5 items }</td>
<td></td>
</tr>
<tr>
<td>usWebConsole</td>
<td>2</td>
<td>[cppmicroservices::HttpServletRequest]</td>
<td>singleton</td>
<td>▶️ { 4 items }</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 10.4: Detailed view on a service interface and registered services implementing it.

10.1. Screenshots
The Shell Service provides an interface to interact with and issue commands to a C++ Micro Services framework. See usShell3 for a client with a textual interface.

**Warning:** This bundle has not reached a stable version yet. Its design and API may change between releases in a backwards incompatible way.
See also Framework for a high-level description.

## 12.1 Command-Line Reference

The following options are supported by the `usResourceCompiler3` program:

---help, -h
   Print usage and exit.

--verbose, -V
   Run in verbose mode.

--bundle-name, -n
   The bundle name as specified in the `US_BUNDLE_NAME` compile definition.

--compression-level, -c
   Compression level used for zip. Value range is 0 to 9. Default value is 6.

--out-file, -o
   Path to output zip file. If the file exists it will be overwritten. If this option is not provided, a temporary zip file will be created.

--res-add, -r
   Path to a resource file, relative to the current working directory.

--zip-add, -z
   Path to a file containing a zip archive to be merged into the output zip file.

--manifest-add, -m
   Path to the bundle’s manifest file. If multiple --manifest-add options are specified, all manifest files will be concatenated into one.

--bundle-file, -b
   Path to the bundle binary. The resources zip file will be appended to this binary.
Note:
1. Only options --res-add, --zip-add and --manifest-add can be specified multiple times.
2. If option --manifest-add or --res-add is specified, option --bundle-name must be provided.
3. At-least one of --bundle-file or --out-file options must be provided.

Hint: If you are using CMake, consider using the provided usFunctionEmbedResources CMake macro which handles the invocation of the usResourceCompiler3 executable and sets up the correct file dependencies. Otherwise, you also need to make sure that the set of static bundles linked into a shared bundle or executable is also in the input file list of your usResourceCompiler3 call for that shared bundle or executable.

Here is a full example creating a bundle and embedding resource data:

```cpp
set(bundle_name "org_me_mybundle")
set(srcs mybundle.cpp)

usFunctionGenerateBundleInit(TARGET mybundle OUT srcs)
usFunctionGetResourceSource(TARGET mybundle OUT srcs)

add_library(mybundle ${srcs})
target_link_libraries(mybundle CppMicroServices)

set_property(TARGET mybundle APPEND PROPERTY COMPILE_DEFINITIONS US_BUNDLE_NAME=${bundle_name})
set_property(TARGET mybundle PROPERTY US_BUNDLE_NAME ${bundle_name})

usFunctionEmbedResources(TARGET mybundle FILES hello.txt)
```

If you are not using CMake, you can run the resource compiler from the command line yourself.

## 12.2 Example usage

Construct a zip blob with contents `mybundle/manifest.json`, merge the contents of zip file `filetomerge.zip` into it and write the resulting blob into `Example.zip`:

```command_line
usResourceCompiler3 --compression-level 9 --verbose --bundle-name mybundle --out-file Example.zip --manifest-add manifest.json --zip-add filetomerge.zip
```

Construct a zip blob with contents `mybundle/manifest.json`, merge the contents of zip file `archivetomerge.zip` into it and append the resulting zip blob to `mybundle.dylib`:

```command_line
usResourceCompiler3 -V -n mybundle -b mybundle.dylib -m manifest.json -z archivetomerge.zip
```

Append the contents of `archivetoembed.zip` to `mybundle.dll`:

```command_line
usResourceCompiler3.exe -b mybundle.dll -z archivetoembed.zip
```
Construct a zip blob with the contents of manifest_part1.json and auto_generated_manifest.json concatenated into mybundle/manifest.json and embed it into mybundle.dll:

```
usResourceCompiler3 -n mybundle -b mybundle.dll -m manifest_part1.json
                 -m auto_generated_manifest.json
```
Warning: The usShell3 is experimental and is subject to change in between minor releases.

13.1 Command-Line Reference

The following options are supported by the usShell3 program:

--help, -h
   Print usage and exit.

--load, -l
   Load bundle.
CMake support for external projects.

External projects can include the CMake scripts provided by the CppMicroServices library to automatically generate bundle initialization code, import static bundles, and to embed external resources into bundles.

14.1 `usFunctionAddResources`

`usFunctionAddResources`

Add resources to a library or executable.

```
usFunctionAddResources(TARGET target [BUNDLE_NAME bundle_name]
    [WORKING_DIRECTORY dir] [COMPRESSION_LEVEL level]
    [FILES res1...] [ZIP_ARCHIVES archive1...])
```

This CMake function uses an external command line program to generate a ZIP archive containing data from external resources such as text files or images or other ZIP archives. The created archive file can be appended or linked into the target file using the `usFunctionEmbedResources()` function.

Each bundle can call this function to add resources and make them available at runtime through the Bundle class. Multiple calls to this function append the input files.

In the case of linking static bundles which contain resources to the target bundle, adding the static bundle target name to the `ZIP_ARCHIVES` list will merge its resources into the target bundle.

Listing 14.1: Example

```
set(bundle_srcs )
usFunctionAddResources(TARGET mylib
    BUNDLE_NAME org_me_mylib
    FILES config.properties logo.png)
```

One-value keywords
• **TARGET** (required): The target to which the resource files are added.

• **BUNDLE_NAME** (required/optional): The bundle name of the target, as specified in the CMake US_BUNDLE_NAME pre-processor definition of that target. This parameter is optional if a target property with the name US_BUNDLE_NAME exists, containing the required bundle name.

• **COMPRESSION_LEVEL** (optional): The zip compression level (0-9). Defaults to the default zip level. Level 0 disables compression.

• **WORKING_DIRECTORY** (optional): The root path for all resource files listed after the FILES argument. If no or a relative path is given, it is considered relative to the current CMake source directory.

**Multi-value keywords**

• **FILES** (optional): A list of resource files (paths to external files in the file system) relative to the current working directory.

• **ZIP_ARCHIVES** (optional): A list of zip archives (relative to the current working directory or absolute file paths) whose contents is merged into the target bundle. If a list entry is a valid target name and that target is a static library, its absolute file path is used instead.

**See also:**

`usFunctionEmbedResources()`

The Resource System

### 14.2 usFunctionEmbedResources

**usFunctionEmbedResources**

Embed resources in a library or executable.

```
usFunctionEmbedResources(TARGET target [BUNDLE_NAME bundle_name] [APPEND | LINK] [WORKING_DIRECTORY dir] [COMPRESSION_LEVEL level] [FILES res1...] [ZIP_ARCHIVES archive1...])
```

This CMake function uses an external command line program to generate a ZIP archive containing data from external resources such as text files or images or other ZIP archives. External resources can be added to a bundle using the `usFunctionAddResources()` function or directly using this function using similar parameters. The created archive file is appended or embedded as a binary blob to the target file.

**Note:** To set-up correct file dependencies from your bundle target to your resource files, you have to add a special source file to the source list of the target. The source file name can be retrieved by using `usFunctionGetResourceSource()`. This ensures that changed resource files will automatically be re-added to the bundle.

There are two different modes for including resources: `APPEND` and `LINK`. In `APPEND` mode, the generated zip file is appended at the end of the target file. In `LINK` mode, the zip file is compiled / linked into the target using platform specific techniques. `LINK` mode is necessary if certain tools make additional assumptions about the object layout of the target file (e.g. codesign on MacOS). `LINK` mode may result in slower bundle initialization and bigger object files. The default mode is `LINK` mode on MacOS and `APPEND` mode on all other platforms.
Listing 14.2: Example

```cpp
usFunctionEmbedResources(TARGET mylib
  BUNDLE_NAME org_me_mylib
  FILES config.properties logo.png
)
```

One-value keywords

- **TARGET** (required): The target to which the resource files are added.
- **BUNDLE_NAME** (required/optional): The bundle name of the target, as specified in the `US_BUNDLE_NAME` pre-processor definition of that target. This parameter is optional if a target property with the name `US_BUNDLE_NAME` exists, containing the required bundle name.

Options

- **APPEND**: Append the resources zip file to the target file.
- **LINK**: Link (embed) the resources zip file if possible.

For the `WORKING_DIRECTORY`, `COMPRESSION_LEVEL`, `FILES`, `ZIP_ARCHIVES` parameters see the documentation of the `usFunctionAddResources` macro which is called with these parameters if set.

See also:

- `usFunctionAddResources()`
- `usFunctionGetResourceSource()`

The Resource System

14.3 usFunctionGetResourceSource

**usFunctionGetResourceSource**

Get a source file name for handling resource dependencies

```cpp
usFunctionGetResourceSource(TARGET target OUT <out_var> [LINK | APPEND])
```

This CMake function retrieves the name of a generated file which has to be added to a bundles source file list to set-up resource file dependencies. This ensures that changed resource files will automatically be re-added to the bundle.

Listing 14.3: Example

```cpp
set(bundle_srcs mylib.cpp)
usFunctionGetResourceSource(TARGET mylib
  OUT bundle_srcs
)
add_library(mylib ${bundle_srcs})
```

One-value keywords

- **TARGET** (required): The name of the target to which the resource files are added.
- **OUT** (required): A list variable to which the file name will be appended.

Options

- **LINK** (optional): Generate a suitable source file for `LINK` mode.
• **APPEND** (optional): Generate a suitable source file for *APPEND* mode.

See also:

```cpp
usFunctionAddResources()
usFunctionEmbedResources()
```

# 14.4 *usFunctionGenerateBundleInit*

**usFunctionGenerateBundleInit**

Generate a source file which handles proper initialization of a bundle.

```
usFunctionGenerateBundleInit(TARGET target OUT <out_var>)
```

This CMake function will append the path to a generated source file to the `out_var` variable, which should be compiled into a bundle. The bundle's source code must be compiled with the `US_BUNDLE_NAME` pre-processor definition.

```
set(bundle_srcs )
usFunctionGenerateBundleInit(TARGET mylib OUT bundle_srcs)
add_library(mylib ${bundle_srcs})
set_property(TARGET ${mylib} APPEND PROPERTY COMPILE_DEFINITIONS US_BUNDLE_NAME=MyBundle)
```

**One-value keywords**

- **TARGET** (required): The name of the target for which the source file is generated.
- **OUT** (required): A list variable to which the path of the generated source file will be appended.
15.1 Main

These classes are the main API to the C++ Micro Services Framework:

15.1.1 Bundle

```cpp
std::ostream & cppmicroservices::operator<<(std::ostream & os, const Bundle &bundle)
Streams a textual representation of bundle into the stream os.
```

```cpp
std::ostream & cppmicroservices::operator<<(std::ostream & os, Bundle const *bundle)
This is the same as calling os << *bundle.
```

```cpp
std::ostream & cppmicroservices::operator<<(std::ostream & os, Bundle::State state)
Streams a textual representation of the bundle state enumeration.
```

```cpp
class cppmicroservices::Bundle
#include <cppmicroservices/Bundle.h> An installed bundle in the Framework.
```

A Bundle object is the access point to define the lifecycle of an installed bundle. Each bundle installed in the CppMicroServices environment has an associated Bundle object.

A bundle has a unique identity, a long, chosen by the Framework. This identity does not change during the lifecycle of a bundle. Uninstalling and then reinstalling the bundle creates a new unique identity.

A bundle can be in one of six states:

- STATE_UNINSTALLED
- STATE_INSTALLED
- STATE_RESOLVED
- STATE_STARTING
- STATE_STOPPING
• **STATE_ACTIVE**

Values assigned to these states have no specified ordering; they represent bit values that may be ORed together to determine if a bundle is in one of the valid states.

A bundle should only have active threads of execution when its state is one of **STATE_STARTING**, **STATE_ACTIVE**, or **STATE_STOPPING**. A **STATE_UNINSTALLED** bundle can not be set to another state; it is a zombie and can only be reached because references are kept somewhere.

The framework is the only entity that is allowed to create **Bundle** objects, and these objects are only valid within the **Framework** that created them.

Bundles have a natural ordering such that if two **Bundle**s have the same **bundle id** they are equal. A **Bundle** is less than another **Bundle** if it has a lower **bundle id** and is greater if it has a higher bundle id.

**Remark** This class is thread safe.

Subclassed by *cppmicroservices::Framework*

**Public Types**

```cpp
enum State
    The bundle state.
    Values:
    STATE_UNINSTALLED = 0x00000001
        The bundle is uninstalled and may not be used.
        The **STATE_UNINSTALLED** state is only visible after a bundle is uninstalled; the bundle is in an unusable state but references to the **Bundle** object may still be available and used for introspection.
        The value of **STATE_UNINSTALLED** is 0x00000001.
    STATE_INSTALLED = 0x00000002
        The bundle is installed but not yet resolved.
        A bundle is in the **STATE_INSTALLED** state when it has been installed in the **Framework** but is not or cannot be resolved.
        This state is visible if the bundle’s code dependencies are not resolved. The **Framework** may attempt to resolve a **STATE_INSTALLED** bundle’s code dependencies and move the bundle to the **STATE_RESOLVED** state.
        The value of **STATE_INSTALLED** is 0x00000002.
    STATE_RESOLVED = 0x00000004
        The bundle is resolved and is able to be started.
        A bundle is in the **STATE_RESOLVED** state when the **Framework** has successfully resolved the bundle’s code dependencies. These dependencies include:
        • None (this may change in future versions)
        Note that the bundle is not active yet. A bundle is put in the **STATE_RESOLVED** state before it can be started. The **Framework** may attempt to resolve a bundle at any time.
        The value of **STATE_RESOLVED** is 0x00000004.
    STATE_STARTING = 0x00000008
        The bundle is in the process of starting.
```
A bundle is in the \texttt{STATE\_STARTING} state when its \texttt{Start} method is active. A bundle must be in this state when the bundle’s \texttt{BundleActivator::Start(BundleContext)} is called. If the \texttt{BundleActivator::Start} method completes without exception, then the bundle has successfully started and moves to the \texttt{STATE\_ACTIVE} state.

If the bundle has a lazy activation policy, then the bundle may remain in this state for some time until the activation is triggered.

The value of \texttt{STATE\_STARTING} is \texttt{0x00000008}.

\texttt{STATE\_STOPPING} = \texttt{0x00000010}
The bundle is in the process of stopping.

A bundle is in the \texttt{STATE\_STOPPING} state when its \texttt{Stop} method is active. A bundle is in this state when the bundle’s \texttt{BundleActivator::Stop(BundleContext)} method is called. When the \texttt{BundleActivator::Stop} method completes the bundle is stopped and moves to the \texttt{STATE\_RESOLVED} state.

The value of \texttt{STATE\_STOPPING} is \texttt{0x00000010}.

\texttt{STATE\_ACTIVE} = \texttt{0x00000020}
The bundle is now running.

A bundle is in the \texttt{STATE\_ACTIVE} state when it has been successfully started and activated.

The value of \texttt{STATE\_ACTIVE} is \texttt{0x00000020}.

\begin{verbatim}
enum StartOptions
    Values:
    START_TRANSIENT = 0x00000001
        The bundle start operation is transient and the persistent autostart setting of the bundle is not modified.
        This bit may be set when calling \texttt{Start(uint32_t)} to notify the framework that the autostart setting of the bundle must not be modified. If this bit is not set, then the autostart setting of the bundle is modified.
        \texttt{See Start(uint32_t)}
        \texttt{Note} This option is reserved for future use and not supported yet.
    START_ACTIVATION_POLICY = 0x00000002
        The bundle start operation must activate the bundle according to the bundle’s declared activation policy.
        This bit may be set when calling \texttt{Start(uint32_t)} to notify the framework that the bundle must be activated using the bundle’s declared activation policy.
        \texttt{See Constants::BUNDLE\_ACTIVATIONPOLICY}
        \texttt{See Start(uint32_t)}
        \texttt{Note} This option is reserved for future use and not supported yet.

enum StopOptions
    Values:
    STOP_TRANSIENT = 0x00000001
        The bundle stop is transient and the persistent autostart setting of the bundle is not modified.
        This bit may be set when calling \texttt{Stop(uint32_t)} to notify the framework that the autostart setting of the bundle must not be modified. If this bit is not set, then the autostart setting of the bundle is modified.
        \texttt{See Stop(uint32_t)}
        \texttt{Note} This option is reserved for future use and not supported yet.

using TimeStamp = std::chrono::steady_clock::time_point
\end{verbatim}
Public Functions

Bundle (const Bundle&)  
Bundle (Bundle&&)  
Bundle &operator=(const Bundle&)  
Bundle &operator=(Bundle&&)

Bundle ()  
Constructs an invalid Bundle object.

Valid bundle objects can only be created by the framework in response to certain calls to a BundleContext object. A BundleContext object is supplied to a bundle via its BundleActivator or as a return value of the GetBundleContext() method.

See operator bool() const

virtual ~Bundle ()

bool operator==(const Bundle &rhs) const  
Compares this Bundle object with the specified bundle.

Valid Bundle objects compare equal if and only if they are installed in the same framework instance and their bundle id is equal. Invalid Bundle objects are always considered to be equal.

Return true if this Bundle object is equal to rhs, false otherwise.

Parameters
• rhs: The Bundle object to compare this object with.

bool operator!=(const Bundle &rhs) const  
Compares this Bundle object with the specified bundle for inequality.

Return Returns the result of !(this == rhs).

Parameters
• rhs: The Bundle object to compare this object with.

bool operator<(const Bundle &rhs) const  
Compares this Bundle with the specified bundle for order.

Bundle objects are ordered first by their framework id and then according to their bundle id. Invalid Bundle objects will always compare greater then valid Bundle objects.

Return

Parameters
• rhs: The Bundle object to compare this object with.

operator bool() const  
Tests this Bundle object for validity.

Invalid Bundle objects are created by the default constructor or can be returned by certain framework methods if the bundle has been uninstalled.
Return `true` if this Bundle object is valid and can safely be used, `false` otherwise.

Bundle &operator=(std::nullptr_t)
Releases any resources held or locked by this Bundle and renders it invalid.

State GetState() const
Returns this bundle’s current state.
A bundle can be in only one state at any time.

Return An element of STATE_UNINSTALLED, STATE_INSTALLED, STATE_RESOLVED,
STATE_STARTING, STATE_STOPPING, STATE_ACTIVE.

Exceptions
- std::invalid_argument: if this bundle is not initialized.

BundleContext GetBundleContext() const
Returns this bundle’s BundleContext.
The returned BundleContext can be used by the caller to act on behalf of this bundle.
If this bundle is not in the STATE_STARTING, STATE_ACTIVE, or STATE_STOPPING states, then
this bundle has no valid BundleContext and this method will return an invalid BundleContext
object.

Return A valid or invalid BundleContext for this bundle.

Exceptions
- std::invalid_argument: if this bundle is not initialized.

long GetBundleId() const
Returns this bundle’s unique identifier.
This bundle is assigned a unique identifier by the framework when it was installed.
A bundle’s unique identifier has the following attributes:
- Is unique.
- Is a long.
- Its value is not reused for another bundle, even after a bundle is uninstalled.
- Does not change while a bundle remains installed.
- Does not change when a bundle is re-started.
This method continues to return this bundle’s unique identifier while this bundle is in the
STATE_UNINSTALLED state.

Return The unique identifier of this bundle.

Exceptions
- std::invalid_argument: if this bundle is not initialized.
std::string GetLocation() const

Returns this bundle’s location.

The location is the full path to the bundle’s shared library. This method continues to return this bundle’s location while this bundle is in the **STATE_UNINSTALLED** state.

**Return** The string representation of this bundle’s location.

**Exceptions**

- std::invalid_argument: if this bundle is not initialized.

void *GetSymbol(void *handle, const std::string &symname) const

Retrieves the resolved symbol from bundle shared library and returns a function pointer associated with it.

**Return** A function pointer to the desired symbol or nullptr if the library is not loaded

**Pre** Bundle is already started and active

**Post** The symbol(s) associated with the bundle gets fetched if the library is loaded

**Post** If the symbol does not exist, the API returns nullptr

**Parameters**

- handle: Handle to the Bundle’s shared library
- symname: Name of the symbol

**Exceptions**

- std::runtime_error: if the bundle is not started or active
- std::invalid_argument: if handle or symname is empty
- std::invalid_argument: if this bundle is not initialized

std::string GetSymbolicName() const

Returns the symbolic name of this bundle as specified by the US_BUNDLE_NAME preprocessor definition.

The bundle symbolic name together with a version must identify a unique bundle.

This method continues to return this bundle’s symbolic name while this bundle is in the **STATE_UNINSTALLED** state.

**Return** The symbolic name of this bundle.

**Exceptions**

- std::invalid_argument: if this bundle is not initialized.

BundleVersion GetVersion() const

Returns the version of this bundle as specified in its manifest.json file.

If this bundle does not have a specified version then BundleVersion::EmptyVersion is returned.

This method continues to return this bundle’s version while this bundle is in the **STATE_UNINSTALLED** state.

**Return** The version of this bundle.

**Exceptions**
• `std::invalid_argument`: if this bundle is not initialized.

```cpp
std::map<std::string, Any> GetProperties() const
    Returns this bundle’s Manifest properties as key/value pairs.
    Deprecated since version 3.0: Use `GetHeaders()` instead.
    See also:
    Bundle Properties
```

Return A map containing this bundle’s Manifest properties as key/value pairs.

Exceptions

```cpp
• `std::invalid_argument`: if this bundle is not initialized.
```

```cpp
const AnyMap & GetHeaders() const
    Returns this bundle’s Manifest headers and values.
    Manifest header names are case-insensitive. The methods of the returned `AnyMap` object operate on
    header names in a case-insensitive manner.
    If a Manifest header value starts with “%”, it is localized according to the default locale. If no localization
    is found for a header value, the header value without the leading “%” is returned.
    This method continues to return Manifest header information while this bundle is in the UNINSTALLED
    state.
    Note Localization is not yet supported, hence the leading “%” is always removed.
    Note The lifetime of the returned reference is bound to the lifetime of this `Bundle` object.
    Return A map containing this bundle’s Manifest headers and values.
    See `Constants::BUNDLE_LOCALIZATION`
    Exceptions
    ```cpp
    • `std::invalid_argument`: if this bundle is not initialized.
```

```cpp
Any GetProperty(const std::string &key) const
    Returns the value of the specified property for this bundle.
    If not found, the framework’s properties are searched. The method returns an empty `Any` if the property is
    not found.
    Deprecated since version 3.0: Use `GetHeaders()` or `BundleContext::GetProperty(const std::string&)` instead.
    See also:
    Bundle Properties
```

Return The value of the requested property, or an empty string if the property is undefined.

Parameters

```cpp
• `key`: The name of the requested property.

Exceptions

15.1. Main
std::vector<std::string> GetPropertyKeys() const
Returns a list of top-level property keys for this bundle.

Deprecated since version 3.0: Use GetHeaders() instead.

See also:
Bundle Properties

Return A list of available property keys.

Exceptions

• std::invalid_argument: if this bundle is not initialized.

std::vector<ServiceReferenceU> GetRegisteredServices() const
Returns this bundle’s ServiceReference list for all services it has registered or an empty list if this bundle
has no registered services.

The list is valid at the time of the call to this method, however, as the framework is a very dynamic
environment, services can be modified or unregistered at anytime.

Return A list of ServiceReference objects for services this bundle has registered.

Exceptions

• std::logic_error: If this bundle has been uninstalled, if the ServiceRegistrationBase object
  is invalid, or if the service is unregistered.
• std::invalid_argument: if this bundle is not initialized.

std::vector<ServiceReferenceU> GetServicesInUse() const
Returns this bundle’s ServiceReference list for all services it is using or returns an empty list if this bundle
is not using any services.

A bundle is considered to be using a service if its use count for that service is greater than zero.

The list is valid at the time of the call to this method, however, as the framework is a very dynamic
environment, services can be modified or unregistered at anytime.

Return A list of ServiceReference objects for all services this bundle is using.

Exceptions

• std::logic_error: If this bundle has been uninstalled, if the ServiceRegistrationBase object
  is invalid, or if the service is unregistered.
• std::invalid_argument: if this bundle is not initialized.

BundleResource GetResource(const std::string &path) const
Returns the resource at the specified path in this bundle.

The specified path is always relative to the root of this bundle and may begin with ‘/’. A path value of
“/” indicates the root of this bundle.

Return A BundleResource object for the given path. If the path cannot be found in this bundle an
invalid BundleResource object is returned.

Parameters
• path: The path name of the resource.

Exceptions
• std::logic_error: If this bundle has been uninstalled.
• std::invalid_argument: if this bundle is not initialized.

std::vector<BundleResource> FindResources(const std::string &path, const std::string &filePattern, bool recurse) const

Returns resources in this bundle.

This method is intended to be used to obtain configuration, setup, localization and other information from this bundle.

This method can either return only resources in the specified path or recurse into subdirectories returning resources in the directory tree beginning at the specified path.

Examples:

```cpp
class Bundle {
public:
    std::vector<BundleResource> FindResources(const std::string &path, const std::string &filePattern, bool recurse) const;
};

auto bundleContext = GetBundleContext();
auto bundle = bundleContext.GetBundle();

// List all XML files in the config directory
std::vector<BundleResource> xmlFiles = bundle.FindResources("config", ".xml", false);

// Find the resource named vertex_shader.txt starting at the root directory
std::vector<BundleResource> shaders = bundle.FindResources("", "vertex_shader.txt", true);
```

Return A vector of BundleResource objects for each matching entry.

Parameters
• path: The path name in which to look. The path is always relative to the root of this bundle and may begin with ‘/’. A path value of “/” indicates the root of this bundle.
• filePattern: The resource name pattern for selecting entries in the specified path. The pattern is only matched against the last element of the resource path. Substring matching is supported using the wildcard character (‘*’). If filePattern is empty, this is equivalent to “*” and matches all resources.
• recurse: If true, recurse into subdirectories. Otherwise only return resources from the specified path.

Exceptions
• std::logic_error: If this bundle has been uninstalled.
• std::invalid_argument: if this bundle is not initialized.

TimeStamp GetLastModified() const

Returns the time when this bundle was last modified.

A bundle is considered to be modified when it is installed, updated or uninstalled.

Return The time when this bundle was last modified.

Exceptions
• std::invalid_argument: if this bundle is not initialized.
void Start(uint32_t options)

Starts this bundle.

If this bundle’s state is `STATE_UNINSTALLED` then a `std::logic_error` is thrown.

The Framework sets this bundle’s persistent autostart setting to `Started with declared activation` if the `START_ACTIVATION_POLICY` option is set or `Started with eager activation` if not set.

The following steps are executed to start this bundle:

1. If this bundle is in the process of being activated or deactivated, then this method waits for activation or deactivation to complete before continuing. If this does not occur in a reasonable time, a `std::runtime_error` is thrown to indicate this bundle was unable to be started.

2. If this bundle’s state is `STATE_ACTIVE`, then this method returns immediately.

3. If the `START_TRANSIENT` option is not set, then set this bundle’s autostart setting to `Started with declared activation` if the `START_ACTIVATION_POLICY` option is set or `Started with eager activation` if not set. When the Framework is restarted and this bundle’s autostart setting is not `Stopped`, this bundle must be automatically started.

4. If this bundle’s state is not `STATE_RESOLVED`, an attempt is made to resolve this bundle. If the Framework cannot resolve this bundle, a `std::runtime_error` is thrown.

5. If the `START_ACTIVATION_POLICY` option is set and this bundle’s declared activation policy is `lazy` then:
   - If this bundle’s state is `STATE_STARTING`, then this method returns immediately.
   - This bundle’s state is set to `STATE_STARTING`.
   - A bundle event of type `BundleEvent#BUNDLE_LAZY_ACTIVATION` is fired.
   - This method returns immediately and the remaining steps will be followed when this bundle’s activation is later triggered.

6. This bundle’s state is set to `STATE_STARTING`.

7. A bundle event of type `BundleEvent#BUNDLE_STARTING` is fired.

8. If the bundle is contained in a shared library, the library is loaded and the `BundleActivator::Start(BundleContext)` method of this bundle’s `BundleActivator` (if one is specified) is called. If the shared library could not be loaded, or the `BundleActivator` is invalid or throws an exception then:
   - This bundle’s state is set to `STATE_STOPPING`.
   - A bundle event of type `BundleEvent#BUNDLE_STOPPING` is fired.
   - Any services registered by this bundle are unregistered.
   - Any services used by this bundle are released.
   - Any listeners registered by this bundle are removed.
   - This bundle’s state is set to `STATE_RESOLVED`.
   - A bundle event of type `BundleEvent#BUNDLE_STOPPED` is fired.
   - A `std::runtime_error` exception is then thrown.

9. If this bundle’s state is `STATE_UNINSTALLED`, because this bundle was uninstalled while the `BundleActivator::Start` method was running, a `std::logic_error` is thrown.

10. This bundle’s state is set to `STATE_ACTIVE`.

11. A bundle event of type `BundleEvent#BUNDLE_STARTED` is fired.
Preconditions

1. `GetState()` in `{ STATE_INSTALLED, STATE_RESOLVED }` or `{ STATE_INSTALLED, STATE_RESOLVED, STATE_STARTING }` if this bundle has a lazy activation policy.

Postconditions, no exceptions thrown

1. Bundle autostart setting is modified unless the `START_TRANSIENT` option was set.
2. `GetState()` in `{ STATE_ACTIVE }` unless the lazy activation policy was used.
3. `BundleActivator::Start()` has been called and did not throw an exception unless the lazy activation policy was used.

Postconditions, when an exception is thrown

1. Depending on when the exception occurred, the bundle autostart setting is modified unless the `START_TRANSIENT` option was set.
2. `GetState()` not in `{ STATE_STARTING, STATE_ACTIVE }`.

Parameters

- `options`: The options for starting this bundle. See `START_TRANSIENT` and `START_ACTIVATION_POLICY`. The Framework ignores unrecognized options.

Exceptions

- `std::runtime_error`: If this bundle could not be started.
- `std::logic_error`: If this bundle has been uninstalled or this bundle tries to change its own state.
- `std::invalid_argument`: if this bundle is not initialized.
- `cppmicroservices::SecurityException`: if the bundle’s shared library to be loaded failed a security check.

```cpp
void Start ()
Starts this bundle with no options.
This method performs the same function as calling `Start(0)`.

See `Start(uint32_t)`

Exceptions

- `std::runtime_error`: If this bundle could not be started.
- `std::logic_error`: If this bundle has been uninstalled or this bundle tries to change its own state.
- `std::invalid_argument`: if this bundle is not initialized.
- `cppmicroservices::SecurityException`: if the bundle’s shared library to be loaded failed a security check.

```cpp
void Stop (uint32_t options)
Stops this bundle.

The following steps are executed when stopping a bundle:

1. If this bundle’s state is `STATE_UNINSTALLED` then a `std::logic_error` is thrown.
2. If this bundle is in the process of being activated or deactivated then this method waits for activation or deactivation to complete before continuing. If this does not occur in a reasonable time, a `std::runtime_error` is thrown to indicate this bundle was unable to be stopped.

3. If the `STOP_TRANSIENT` option is not set then set this bundle’s persistent autostart setting to `Stopped`. When the Framework is restarted and this bundle’s autostart setting is `Stopped`, this bundle will not be automatically started.

4. If this bundle’s state is not `STATE_STARTING` or `STATE_ACTIVE` then this method returns immediately.

5. This bundle’s state is set to `STATE_STOPPING`.

6. A bundle event of type `BundleEvent#BUNDLE_STOPPING` is fired.

7. If this bundle’s state was `STATE_ACTIVE` prior to setting the state to `STATE_STOPPING`, the `BundleActivator#Stop(BundleContext)` method of this bundle’s `BundleActivator`, if one is specified, is called. If that method throws an exception, this method continues to stop this bundle and a `std::runtime_error` is thrown after completion of the remaining steps.

8. Any services registered by this bundle are unregistered.

9. Any services used by this bundle are released.

10. Any listeners registered by this bundle are removed.

11. If this bundle’s state is `STATE_UNINSTALLED`, because this bundle was uninstalled while the `BundleActivator::Stop` method was running, a `std::runtime_error` is thrown.

12. This bundle’s state is set to `STATE_RESOLVED`.

13. A bundle event of type `BundleEvent#BUNDLE_STOPPED` is fired.

Preconditions

1. `GetState()` in `{ STATE_ACTIVE }`.

Postconditions, no exceptions thrown

1. Bundle autostart setting is modified unless the `STOP_TRANSIENT` option was set.

2. `GetState()` not in `{ STATE_ACTIVE, STATE_STOPPING }`.

3. `BundleActivator::Stop` has been called and did not throw an exception.

Postconditions, when an exception is thrown

1. Bundle autostart setting is modified unless the `STOP_TRANSIENT` option was set.

Parameters

- options: The options for stopping this bundle. See `STOP_TRANSIENT`. The Framework ignores unrecognized options.

Exceptions

- `std::runtime_error`: If the bundle failed to stop.

- `std::logic_error`: If this bundle has been uninstalled or this bundle tries to change its own state.

- `std::invalid_argument`: If this bundle is not initialized.
void Stop()
    Stops this bundle with no options.
    This method performs the same function as calling Stop(0).

See Stop(uint32_t)

Exceptions
    • std::runtime_error: If the bundle failed to stop.
    • std::logic_error: If this bundle has been uninstalled or this bundle tries to change its own state.

void Uninstall()
    Uninstalls this bundle.
    This method causes the Framework to notify other bundles that this bundle is being uninstalled, and then puts this bundle into the STATE_UNINSTALLED state. The Framework removes any resources related to this bundle that it is able to remove.

The following steps are executed to uninstall a bundle:
1. If this bundle’s state is STATE_UNINSTALLED, then a std::logic_error is thrown.
2. If this bundle’s state is STATE_ACTIVE, STATE_STARTING or STATE_STOPPING, this bundle is stopped as described in the Bundle::Stop method. If Bundle::Stop throws an exception, a Framework event of type FrameworkEvent#FRAMEWORK_ERROR is fired containing the exception.
3. This bundle’s state is set to STATE_UNINSTALLED.
4. A bundle event of type BundleEvent#BUNDLE_UNINSTALLED is fired.
5. This bundle and any persistent storage area provided for this bundle by the Framework are removed.

Preconditions
    • GetState() not in { STATE_UNINSTALLED }.

Postconditions, no exceptions thrown
    • GetState() in { STATE_UNINSTALLED }.
    • This bundle has been uninstalled.

Postconditions, when an exception is thrown
    • GetState() not in { STATE_UNINSTALLED }.
    • This Bundle has not been uninstalled.

See Stop()

Exceptions
    • std::runtime_error: If the uninstall failed. This can occur if another thread is attempting to change this bundle’s state and does not complete in a timely manner.
    • std::logic_error: If this bundle has been uninstalled or this bundle tries to change its own state.
    • std::invalid_argument: if this bundle is not initialized.
Protected Functions

Bundle (const std::shared_ptr<BundlePrivate> &d)

Protected Attributes

std::shared_ptr<BundlePrivate> d
std::shared_ptr<CoreBundleContext> c

Friends

friend gr_bundle::BundleRegistry

Bundle MakeBundle (const std::shared_ptr<BundlePrivate>&)

15.1.2 BundleActivator

struct cppmicroservices::BundleActivator
Customizes the starting and stopping of a CppMicroServices bundle.

BundleActivator is an interface that can be implemented by CppMicroServices bundles. The CppMicroServices library can create instances of a bundle's BundleActivator as required. If an instance's BundleActivator::Start method executes successfully, it is guaranteed that the same instance's BundleActivator::Stop method will be called when the bundle is to be stopped. The CppMicroServices library does not concurrently call a BundleActivator object.

BundleActivator is an abstract class interface whose implementations must be exported via a special macro. Implementations are usually declared and defined directly in .cpp files.

class MyActivator : public BundleActivator
{
  public:
    void Start(BundleContext /*context*/)  
    {  /* register stuff */  }

    void Stop(BundleContext /*context*/)  
    {  /* cleanup */  }
};

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(MyActivator)

The class implementing the BundleActivator interface must have a public default constructor so that a BundleActivator object can be created by the CppMicroServices library.

Note: A bundle activator needs to be exported by using the CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR macro. The bundle manifest.json resource also needs to contain a
Public Functions

virtual ~BundleActivator()

virtual void Start(BundleContext context) = 0
   Called when this bundle is started.
   
   This method can be used to register services or to allocate any resources that this bundle may need globally (during the whole bundle lifetime).
   
   This method must complete and return to its caller in a timely manner.

   Parameters
   
   • context: The execution context of the bundle being started.

   Exceptions
   
   • std::exception: If this method throws an exception, this bundle is marked as stopped and the framework will remove this bundle’s listeners, unregister all services registered by this bundle, and release all services used by this bundle.

virtual void Stop(BundleContext context) = 0
   Called when this bundle is stopped.
   
   In general, this method should undo the work that the BundleActivator::Start method started.
   There should be no active threads that were started by this bundle when this method returns.
   
   This method must complete and return to its caller in a timely manner.

   Parameters
   
   • context: The execution context of the bundle being stopped.

   Exceptions
   
   • std::exception: If this method throws an exception, the bundle is still marked as stopped, and the framework will remove the bundle’s listeners, unregister all services registered by the bundle, and release all services used by the bundle.

15.1.3 BundleContext

class cppmicroservices::BundleContext
   A bundle’s execution context within the framework.

   The context is used to grant access to other methods so that this bundle can interact with the framework.

   BundleContext methods allow a bundle to:
   
   • Install other bundles.
   • Subscribe to events published by the framework.
   • Register service objects with the framework service registry.
• Retrieve ServiceReferences from the framework service registry.
• Get and release service objects for a referenced service.
• Get the list of bundles installed in the framework.
• Get the Bundle object for a bundle.

A BundleContext object will be created and provided to the bundle associated with this context when it is started using the BundleActivator::Start method. The same BundleContext object will be passed to the bundle associated with this context when it is stopped using the BundleActivator::Stop method. A BundleContext object is generally for the private use of its associated bundle and is not meant to be shared with other bundles in the bundle environment.

The Bundle object associated with a BundleContext object is called the context bundle.

The BundleContext object is only valid during the execution of its context bundle; that is, during the period when the context bundle is started. If the BundleContext object is used subsequently, a std::runtime_error is thrown. The BundleContext object is never reused after its context bundle is stopped.

The framework is the only entity that can create BundleContext objects.

Remark This class is thread safe.

Public Functions

BundleContext ()
Constructs an invalid BundleContext object.

Valid bundle context objects can only be created by the framework and are supplied to a bundle via its BundleActivator or as a return value of the GetBundleContext () method.

See operator bool() const

bool operator== (const BundleContext &rhs) const
Compares this BundleContext object with the specified bundle context.

Valid BundleContext objects are equal if and only if they represent the same context. Invalid BundleContext objects are always considered to be equal.

Return true if this BundleContext object is equal to rhs, false otherwise.

Parameters
• rhs: The BundleContext object to compare this object with.

bool operator!= (const BundleContext &rhs) const
Compares this BundleContext object with the specified bundle context for inequality.

Return Returns the result of !(this == rhs).

Parameters
• rhs: The BundleContext object to compare this object with.
bool operator< (const BundleContext &rhs) const
Compress this BundleContext with the specified bundle context for order.

How valid BundleContext objects are ordered is an implementation detail and must not be relied on. Invalid BundleContext objects will always compare greater then valid BundleContext objects.

Return true if this object is ordered before rhs, false otherwise.

Parameters
- rhs: The BundleContext object to compare this object with.

operator bool () const
Tests this BundleContext object for validity.
Invalid BundleContext objects are created by the default constructor or can be returned by certain framework methods if the context bundle has been uninstalled.
A BundleContext object can become invalid by assigning a nullptr to it or if the context bundle is stopped.

Return true if this BundleContext object is valid and can safely be used, false otherwise.

BundleContext &operator= (std::nullptr_t)
Releases any resources held or locked by this BundleContext and renders it invalid.

Any GetProperty (const std::string &key) const
Returns the value of the specified property.
If the key is not found in the Framework properties, the method returns an empty Any.

Return The value of the requested property, or an empty Any if the property is undefined.

Parameters
- key: The name of the requested property.

Exceptions
- std::runtime_error: If this BundleContext is no longer valid.

AnyMap GetProperties () const
Returns all known properties.

Return A map of all framework properties.

Exceptions
- std::runtime_error: If this BundleContext is no longer valid.

Bundle GetBundle () const
Returns the Bundle object associated with this BundleContext.
This bundle is called the context bundle.

Return The Bundle object associated with this BundleContext.
Exceptions
- std::runtime_error: If this BundleContext is no longer valid.
**Bundle GetBundle** (long id) const

Returns the bundle with the specified identifier.

**Return** A Bundle object or nullptr if the identifier does not match any previously installed bundle.

**Parameters**

- id: The identifier of the bundle to retrieve.

**Exceptions**

- std::logic_error: If the framework instance is not active.
- std::runtime_error: If this BundleContext is no longer valid.

std::vector<Bundle> GetBundles (const std::string &location) const

Get the bundles with the specified bundle location.

**Return** The requested Bundle(s) or an empty list.

**Parameters**

- location: The location of the bundles to get.

**Exceptions**

- std::logic_error: If the framework instance is not active.
- std::runtime_error: If the BundleContext is no longer valid.

std::vector<Bundle> GetBundles () const

Returns a list of all known bundles.

This method returns a list of all bundles installed in the bundle environment at the time of the call to this method. This list will also contain bundles which might already have been stopped.

**Return** A std::vector of Bundle objects which will hold one object per known bundle.

**Exceptions**

- std::runtime_error: If the BundleContext is no longer valid.

ServiceRegistrationU RegisterService (const InterfaceMapConstPtr &service, const ServiceProperties &properties = ServiceProperties ())

Registers the specified service object with the specified properties under the specified class names into the framework.

A ServiceRegistration object is returned. The ServiceRegistration object is for the private use of the bundle registering the service and should not be shared with other bundles. The registering bundle is defined to be the context bundle. Other bundles can locate the service by using either the GetServiceReferences or GetServiceReference method.

A bundle can register a service object that implements the ServiceFactory or PrototypeServiceFactory interface to have more flexibility in providing service objects to other bundles.

The following steps are taken when registering a service:

1. The framework adds the following service properties to the service properties from the specified ServiceProperties (which may be omitted): A property named Constants::SERVICE_ID identifying the registration number of the service A property named Constants::OBJECTCLASS containing all the specified classes. A property named Constants::SERVICE_SCOPE identifying the scope of the service. Properties with these names in the specified ServiceProperties will be ignored.
2. The service is added to the framework service registry and may now be used by other bundles.
3. A service event of type ServiceEvent::SERVICE_REGISTERED is fired.
4. A ServiceRegistration object for this registration is returned.

Note This is a low-level method and should normally not be used directly. Use one of the templated RegisterService methods instead.

Return A ServiceRegistration object for use by the bundle registering the service to update the service’s properties or to unregister the service.

See ServiceRegistration
See ServiceFactory
See PrototypeServiceFactory

Parameters

• service: A shared_ptr to a map of interface identifiers to service objects.
• properties: The properties for this service. The keys in the properties object must all be std::string objects. See Constants for a list of standard service property keys. Changes should not be made to this object after calling this method. To update the service’s properties the ServiceRegistration::SetProperties method must be called. The set of properties may be omitted if the service has no properties.

Exceptions

• std::runtime_error: If this BundleContext is no longer valid, or if there are case variants of the same key in the supplied properties map.
• std::invalid_argument: If the InterfaceMap is empty, or if a service is registered as a null class.

template <class I1, class... Interfaces, class Impl>
ServiceRegistration<I1, Interfaces...> RegisterService (const std::shared_ptr<Impl> &impl, const ServiceProperties &properties = ServiceProperties())

Registers the specified service object with the specified properties using the specified interfaces types with the framework.

This method is provided as a convenience when registering a service under two interface classes whose type is available to the caller. It is otherwise identical to RegisterService(const InterfaceMap&, const ServiceProperties&) but should be preferred since it avoids errors in the string literal identifying the class name or interface identifier.

Example usage:

```cpp
class MyService2
 : public InterfaceA
 , public InterfaceB
{};

std::shared_ptr<MyService2> myService = std::make_shared<MyService2>();
context.RegisterService<InterfaceA, InterfaceB>(myService);
```

Return A ServiceRegistration object for use by the bundle registering the service to update the service’s properties or to unregister the service.

See RegisterService(const InterfaceMap&, const ServiceProperties&)
Template Parameters

- **I1**: The first interface type under which the service can be located.
- **Interfaces**: Additional interface types under which the service can be located.

Parameters

- **impl**: A shared_ptr to the service object
- **properties**: The properties for this service.

Exceptions

- **std::logic_error**: If this BundleContext is no longer valid.
- **ServiceException**: If the service type S is invalid or the service object is nullptr.

```cpp
template <class I1, class... Interfaces>
ServiceRegistration<I1, Interfaces...> RegisterService(const std::shared_ptr<ServiceFactory>& factory, const ServiceProperties& properties = ServiceProperties())
```

Registers the specified service factory as a service with the specified properties using the specified template argument as service interface type with the framework.

This method is provided as a convenience when `factory` will only be registered under a single class name whose type is available to the caller. It is otherwise identical to `RegisterService(const InterfaceMap&, const ServiceProperties&)` but should be preferred since it avoids errors in the string literal identifying the class name or interface identifier.

Example usage:

```cpp
class MyService2 : public InterfaceA, public InterfaceB {};

class MyServiceFactory : public ServiceFactory
{
  virtual InterfaceMapConstPtr GetService(
    const Bundle& /*bundle*/,
    const ServiceRegistrationBase& /*registration*/)
  {
    return MakeInterfaceMap<InterfaceA, InterfaceB>(
      std::make_shared<MyService2>();
  }

  virtual void UngetService(const Bundle& /*bundle*/,
    const ServiceRegistrationBase& /*registration*/,
    const InterfaceMapConstPtr& /*service*/)
  {
  }
};

std::shared_ptr<MyServiceFactory> myServiceFactory = std::make_shared<MyServiceFactory>();
context.RegisterService<InterfaceA, InterfaceB>(
  ToFactory(myServiceFactory));
```

Return A `ServiceRegistration` object for use by the bundle registering the service to update the service’s properties or to unregister the service.
See `RegisterService(const InterfaceMap&, const ServiceProperties&)`

### Template Parameters

- **I1**: The first interface type under which the service can be located.
- **Interfaces**: Additional interface types under which the service can be located.

### Parameters

- **factory**: A `shared_ptr` to the `ServiceFactory` object.
- **properties**: The properties for this service.

### Exceptions

- `std::logic_error`: If this `BundleContext` is no longer valid.
- `ServiceException`: If the service type `S` is invalid or the service factory object is `nullptr`.

```cpp
std::vector<ServiceReferenceU> GetServiceReferences(const std::string &clazz, const std::string &filter = std::string())
```

Returns a list of `ServiceReference` objects.

The returned list contains services that were registered under the specified class and match the specified filter expression.

The list is valid at the time of the call to this method. However, since the framework is a very dynamic environment, services can be modified or unregistered at any time.

The specified filter expression is used to select the registered services whose service properties contain keys and values that satisfy the filter expression. See `LDAPFilter` for a description of the filter syntax. If the specified filter is empty, all registered services are considered to match the filter. If the specified filter expression cannot be parsed, an `std::invalid_argument` will be thrown with a human-readable message where the filter became unparsable.

The result is a list of `ServiceReference` objects for all services that meet all of the following conditions:

- If the specified class name, `clazz`, is not empty, the service must have been registered with the specified class name. The complete list of class names with which a service was registered is available from the service’s `objectClass` property.
- If the specified `filter` is not empty, the filter expression must match the service.

**Return** A list of `ServiceReference` objects or an empty list if no services are registered that satisfy the search.

### Parameters

- **clazz**: The class name with which the service was registered or an empty string for all services.
- **filter**: The filter expression or empty for all services.

### Exceptions

- `std::invalid_argument`: If the specified `filter` contains an invalid filter expression that cannot be parsed.
- `std::runtime_error`: If this `BundleContext` is no longer valid.
- `std::logic_error`: If the `ServiceRegistrationBase` object is invalid, or if the service is unregistered.

```cpp
template <class S>
std::vector<ServiceReference<S>> GetServiceReferences(const std::string &filter = std::string())

Returns a list of ServiceReference objects.

The returned list contains services that were registered under the interface id of the template argument S and match the specified filter expression.

This method is identical to GetServiceReferences(const std::string&, const std::string&) except that the class name for the service object is automatically deduced from the template argument.

Return A list of ServiceReference objects or an empty list if no services are registered which satisfy the search.

See GetServiceReferences(const std::string&, const std::string&)

Template Parameters

- S: The type under which the requested service objects must have been registered.

Parameters

- filter: The filter expression or empty for all services.

Exceptions

- std::invalid_argument: If the specified filter contains an invalid filter expression that cannot be parsed.
- std::runtime_error: If this BundleContext is no longer valid.
- ServiceException: If the service interface id of S is empty, see Service Interface.

ServiceReferenceU GetServiceReference (const std::string &clazz)

Returns a ServiceReference object for a service that implements and was registered under the specified class.

The returned ServiceReference object is valid at the time of the call to this method. However as the Micro Services framework is a very dynamic environment, services can be modified or unregistered at any time.

This method is the same as calling BundleContext::GetServiceReferences(const std::string&, const std::string&) with an empty filter expression. It is provided as a convenience for when the caller is interested in any service that implements the specified class.

If multiple such services exist, the service with the highest ranking (as specified in its Constants::SERVICE_RANKING property) is returned.

If there is a tie in ranking, the service with the lowest service ID (as specified in its Constants::SERVICE_ID property); that is, the service that was registered first is returned.

Return A ServiceReference object, or an invalid ServiceReference if no services are registered which implement the named class.

See GetServiceReferences(const std::string&, const std::string&)

Parameters

- clazz: The class name with which the service was registered.

Exceptions

- std::runtime_error: If this BundleContext is no longer valid.
ServiceReference<S> GetServiceReference()  
Returns a ServiceReference object for a service that implements and was registered under the specified template class argument.

This method is identical to GetServiceReference(const std::string&) except that the class name for the service object is automatically deduced from the template argument.

Return A ServiceReference object, or an invalid ServiceReference if no services are registered which implement the type S.

See GetServiceReference(const std::string&)
See GetServiceReferences(const std::string&)

Template Parameters
• S: The type under which the requested service must have been registered.

Exceptions
• std::runtime_error: If this BundleContext is no longer valid.
• ServiceException: If the service interface id of S is empty, see Service Interface.

std::shared_ptr<void> GetService(const ServiceReferenceBase& reference)  
Returns the service object referenced by the specified ServiceReferenceBase object.

A bundle’s use of a service is tracked by the bundle’s use count of that service. Each call to GetService(const ServiceReferenceBase& S) increments the context bundle’s use count by one. The deleter function of the returned shared_ptr object is responsible for decrementing the context bundle’s use count.

When a bundle’s use count for a service drops to zero, the bundle should no longer use that service.

This method will always return an empty object when the service associated with this reference has been unregistered.

The ServiceObjects object must be used to obtain multiple service objects for services with prototype scope. For services with singleton or bundle scope, the ServiceObjects::GetService() method behaves the same as the GetService(const ServiceReferenceBase& S) method. That is, only one, use-counted service object is available from the ServiceObjects object.

The following steps are taken to get the service object:
1. If the service has been unregistered, an empty object is returned.
2. The context bundle’s use count for this service is incremented by one.
3. If the context bundle’s use count for the service is currently one and the service was registered with an object implementing the ServiceFactory interface, the ServiceFactory::GetService method is called to create a service object for the context bundle. This service object is cached by the framework. While the context bundle’s use count for the service is greater than zero, subsequent calls to get the services’s service object for the context bundle will return the cached service object. If the ServiceFactory object throws an exception, empty object is returned and a warning is logged.
4. A shared_ptr to the service object is returned.

Return A shared_ptr to the service object associated with reference. An empty shared_ptr is returned if the service is not registered or the ServiceFactory threw an exception.

See ServiceFactory
See ServiceObjects

Parameters
- **reference**: A reference to the service.

**Exceptions**

- `std::runtime_error`: If this `BundleContext` is no longer valid.
- `std::invalid_argument`: If the specified `ServiceReferenceBase` is invalid (default constructed).
- `cppmicroservices::SecurityException`: if retrieving a service caused a bundle’s shared library to be loaded and the bundle failed a security check.

---

```cpp
InterfaceMapConstPtr GetService (const ServiceReferenceU &reference)
template <class S>
std::shared_ptr<S> GetService (const ServiceReference<S> &reference)
```

Returns the service object referenced by the specified `ServiceReference` object.

This is a convenience method which is identical to `void* GetService(const ServiceReferenceBase&)` except that it casts the service object to the supplied template argument type.

**Return** A shared_ptr to the service object associated with `reference`. An empty object is returned if the service is not registered, the `ServiceFactory` threw an exception or the service could not be cast to the desired type.

**See** `GetService(const ServiceReferenceBase&)`

**See** `ServiceFactory`

**Template Parameters**

- **S**: The type the service object will be cast to.

**Exceptions**

- `std::runtime_error`: If this `BundleContext` is no longer valid.
- `std::invalid_argument`: If the specified `ServiceReference` is invalid (default constructed).
- `cppmicroservices::SecurityException`: if retrieving a service caused a bundle’s shared library to be loaded and the bundle failed a security check.

```cpp
template <class S>
ServiceObjects<S> GetServiceObjects (const ServiceReference<S> &reference)
```

Returns the `ServiceObjects` object for the service referenced by the specified `ServiceReference` object.

The `ServiceObjects` object can be used to obtain multiple service objects for services with prototype scope. For services with singleton or bundle scope, the `ServiceObjects::getService()` method behaves the same as the `GetService(const ServiceReference<S>&)` method. That is, only one, use-counted service object is available from the `ServiceObjects` object.

**Return** A `ServiceObjects` object for the service associated with the specified reference or an invalid instance if the service is not registered.

**See** `PrototypeServiceFactory`

**Template Parameters**

- **S**: Type of Service.

**Parameters**

- `reference`: A reference to the service.
Exceptions

- `std::runtime_error`: If this `BundleContext` is no longer valid.
- `std::invalid_argument`: If the specified `ServiceReference` is invalid (default constructed or the service has been unregistered)
- `cppmicroservices::SecurityException`: if retrieving a service caused a bundle’s shared library to be loaded and the bundle failed a security check.

ListenerToken `AddServiceListener` (const ServiceListener &`listener`, const std::string &`filter` = std::string())

Adds the specified `listener` with the specified `filter` to the context bundle’s list of listeners.

See `LDAPFilter` for a description of the filter syntax. Listeners are notified when a service has a lifecycle state change.

The framework takes care of removing all listeners registered by this context bundle’s classes after the bundle is stopped.

The `listener` is called if the filter criteria is met. To filter based upon the class of the service, the filter should reference the `Constants::OBJECTCLASS` property. If `filter` is empty, all services are considered to match the filter.

When using a `filter`, it is possible that the `ServiceEvents` for the complete lifecycle of a service will not be delivered to the `listener`. For example, if the `filter` only matches when the property `example_property` has the value 1, the `listener` will not be called if the service is registered with the property `example_property` not set to the value 1. Subsequently, when the service is modified setting property `example_property` to the value 1, the filter will match and the `listener` will be called with a `ServiceEvent` of type `SERVICE_MODIFIED`. Thus, the `listener` will not be called with a `ServiceEvent` of type `SERVICE_REGISTERED`.

Return a ListenerToken object which can be used to remove the `listener` from the list of registered listeners.

See `ServiceEvent`

See `ServiceListener`

See `RemoveServiceListener()`

Parameters

- `listener`: Any callable object.
- `filter`: The filter criteria.

Exceptions

- `std::invalid_argument`: If `filter` contains an invalid filter string that cannot be parsed.
- `std::runtime_error`: If this `BundleContext` is no longer valid.

void `RemoveServiceListener` (const ServiceListener &`listener`)

Removes the specified `listener` from the context bundle’s list of listeners.

If the `listener` is not contained in this context bundle’s list of listeners, this method does nothing.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `RemoveListener()` instead.

See `AddServiceListener()`
Parameters

- **listener**: The callable object to remove.

Exceptions

- **std::runtime_error**: If this `BundleContext` is no longer valid.

ListenerToken **AddBundleListener**(const BundleListener &`listener`)

Adds the specified `listener` to the context bundle’s list of listeners.

Listeners are notified when a bundle has a lifecycle state change.

**Return** a ListenerToken object which can be used to remove the `listener` from the list of registered listeners.

See **BundleEvent**

See **BundleListener**

Parameters

- **listener**: Any callable object.

Exceptions

- **std::runtime_error**: If this `BundleContext` is no longer valid.

void **RemoveBundleListener**(const BundleListener &`listener`)

Removes the specified `listener` from the context bundle’s list of listeners.

If the `listener` is not contained in this context bundle’s list of listeners, this method does nothing.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `RemoveListener()` instead.

See **AddBundleListener()**

See **BundleListener**

Parameters

- **listener**: The callable object to remove.

Exceptions

- **std::runtime_error**: If this `BundleContext` is no longer valid.

ListenerToken **AddFrameworkListener**(const FrameworkListener &`listener`)

Adds the specified `listener` to the context bundle’s list of framework listeners.

Listeners are notified of framework events.

**Return** a ListenerToken object which can be used to remove the `listener` from the list of registered listeners.

See **FrameworkEvent**

See **FrameworkListener**

Parameters

- **listener**: Any callable object.

Exceptions

- **std::runtime_error**: If this `BundleContext` is no longer valid.
std::runtime_error: If this BundleContext is no longer valid.

```cpp
void RemoveFrameworkListener (const FrameworkListener &listener)
Removes the specified listener from the context bundle’s list of framework listeners.
If the listener is not contained in this context bundle’s list of listeners, this method does nothing.
Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be
removed in the next major release. Use RemoveListener() instead.

See AddFrameworkListener()
See FrameworkListener
Parameters
• listener: The callable object to remove.
Exceptions
• std::runtime_error: If this BundleContext is no longer valid.
```

```cpp
void RemoveListener (ListenerToken token)
Removes the registered listener associated with the token
If the listener associated with the token is not contained in this context bundle’s list of listeners or if
token is an invalid token, this method does nothing.
The token can correspond to one of Service, Bundle or Framework listeners. Using this function to remove
the registered listeners is the recommended approach over using any of the other deprecated functions -
Remove{Bundle, Framework, Service}Listener.

See AddServiceListener()
See AddBundleListener()
See AddFrameworkListener()
Parameters
• token: is an object of type ListenerToken.
Exceptions
• std::runtime_error: If this BundleContext is no longer valid.
```

```cpp
template <class R>
ListenerToken cppmicroservices::BundleContext::AddServiceListener(R * receiver, void(R::*)(const ServiceEvent &) callback, const std::string & filter = std::string())
Adds the specified callback with the specified filter to the context bundles’s list of listeners.
See LDAPFilter for a description of the filter syntax. Listeners are notified when a service has a lifecycle
state change.
You must take care to remove registered listeners before the receiver object is destroyed. However, the
Micro Services framework takes care of removing all listeners registered by this context bundle’s classes
after the bundle is stopped.
If the context bundle’s list of listeners already contains a pair (r, c) of receiver and callback
such that (r == receiver && c == callback), then this method replaces that callback’s filter
(which may be empty) with the specified one (which may be empty).
```
The callback is called if the filter criteria is met. To filter based upon the class of the service, the filter should reference the `Constants::OBJECTCLASS` property. If `filter` is empty, all services are considered to match the filter.

When using a `filter`, it is possible that the `ServiceEvents` for the complete lifecycle of a service will not be delivered to the callback. For example, if the `filter` only matches when the property `example_property` has the value 1, the callback will not be called if the service is registered with the property `example_property` not set to the value 1. Subsequently, when the service is modified setting property `example_property` to the value 1, the filter will match and the callback will be called with a `ServiceEvent` of type `SERVICE_MODIFIED`. Thus, the callback will not be called with a `ServiceEvent` of type `SERVICE_REGISTERED`.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `std::bind` to bind the member function and then pass the result to `AddServiceListener(const ServiceListener&)` instead.

**Return** a `ListenerToken` object which can be used to remove the callable from the registered listeners.

**See** `ServiceEvent`

**See** `RemoveServiceListener()`

**Template Parameters**
- `R`: The type of the receiver (containing the member function to be called)

**Parameters**
- `receiver`: The object to connect to.
- `callback`: The member function pointer to call.
- `filter`: The filter criteria.

**Exceptions**
- `std::invalid_argument`: If `filter` contains an invalid filter string that cannot be parsed.
- `std::runtime_error`: If this `BundleContext` is no longer valid.

```cpp
template <class R>
void cppmicroservices::BundleContext::RemoveServiceListener(R * receiver, void(R::*)(const ServiceEvent & callback))
```

Removes the specified callback from the context bundle’s list of listeners.

If the (receiver, callback) pair is not contained in this context bundle’s list of listeners, this method does nothing.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `RemoveListener()` instead.

**See** `AddServiceListener()`

**Template Parameters**
- `R`: The type of the receiver (containing the member function to be removed)

**Parameters**
- `receiver`: The object from which to disconnect.
- `callback`: The member function pointer to remove.

**Exceptions**
- `std::runtime_error`: If this `BundleContext` is no longer valid.
template <class R>
ListenerToken cppmicroservices::BundleContext::AddBundleListener(R * receiver, void(R::*)(const BundleEvent &) callback)

Adds the specified callback to the context bundle's list of listeners.

Listeners are notified when a bundle has a lifecycle state change.

If the context bundle's list of listeners already contains a pair \((r, c)\) of receiver and callback such that \((r == receiver && c == callback)\), then this method does nothing.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `std::bind` to bind the member function and then pass the result to `AddBundleListener(const BundleListener&)` instead.

Return a ListenerToken object which can be used to remove the callable from the registered listeners.

See `BundleEvent`

Template Parameters
• R: The type of the receiver (containing the member function to be called)

Parameters
• receiver: The object to connect to.
• callback: The member function pointer to call.

Exceptions
• std::runtime_error: If this `BundleContext` is no longer valid.

template <class R>
void cppmicroservices::BundleContext::RemoveBundleListener(R * receiver, void(R::*)(const BundleEvent &) callback)

Removes the specified callback from the context bundle's list of listeners.

If the \((receiver, callback)\) pair is not contained in this context bundle's list of listeners, this method does nothing.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `RemoveListener()` instead.

See `AddBundleListener()`

Template Parameters
• R: The type of the receiver (containing the member function to be removed)

Parameters
• receiver: The object from which to disconnect.
• callback: The member function pointer to remove.

Exceptions
• std::runtime_error: If this `BundleContext` is no longer valid.

template <class R>
ListenerToken cppmicroservices::BundleContext::AddFrameworkListener(R * receiver, void(R::*)(const FrameworkEvent &) callback)

Adds the specified callback to the context bundle's list of framework listeners.

Listeners are notified of framework events.

If the context bundle's list of listeners already contains a pair \((r, c)\) of receiver and callback such that \((r == receiver && c == callback)\), then this method does nothing.
Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `std::bind` to bind the member function and then pass the result to `AddFrameworkListener(const FrameworkListener&)` instead.

**Return** a ListenerToken object which can be used to remove the callable from the registered listeners.

See `FrameworkEvent`

**Template Parameters**
- `R`: The type of the receiver (containing the member function to be called)

**Parameters**
- `receiver`: The object to connect to.
- `callback`: The member function pointer to call.

**Exceptions**
- `std::runtime_error`: If this `BundleContext` is no longer valid.

```
template <class R>
void cppmicroservices::BundleContext::RemoveFrameworkListener(R * receiver, void(R::*)(const FrameworkEvent &))
```

Removes the specified `callback` from the context bundle’s list of framework listeners.

If the `(receiver, callback)` pair is not contained in this context bundle’s list of listeners, this method does nothing.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `RemoveListener()` instead.

See `AddFrameworkListener()`

**Template Parameters**
- `R`: The type of the receiver (containing the member function to be removed)

**Parameters**
- `receiver`: The object from which to disconnect.
- `callback`: The member function pointer to remove.

**Exceptions**
- `std::runtime_error`: If this `BundleContext` is no longer valid.

```
std::string GetDataFile(const std::string & filename) const
```

Get the absolute path for a file or directory in the persistent storage area provided for the bundle.

The absolute path for the base directory of the persistent storage area provided for the context bundle by the `Framework` can be obtained by calling this method with an empty string as `filename`.

**Return** The absolute path to the persistent storage area for the given file name.

**Parameters**
- `filename`: A relative name to the file or directory to be accessed.

**Exceptions**
- `std::runtime_error`: If this `BundleContext` is no longer valid.
- `std::invalid_argument`: If the input param `filename` is not a valid UTF-8 string.
std::vector<Bundle> InstallBundles(const std::string &location, const cppmicroservices::AnyMap &bundleManifest = cppmicroservices::AnyMap (cppmicroservices::any_map::UNORDERED_MAP_CASEINSENSITIVE_KEYS))

Installs all bundles from the bundle library at the specified location.

The following steps are required to install a bundle:

1. If a bundle containing the same install location is already installed, the Bundle object for that bundle is returned.

2. The bundle’s associated resources are allocated. The associated resources minimally consist of a unique identifier and a persistent storage area if the platform has file system support. If this step fails, a std::runtime_error is thrown.

3. A bundle event of type BundleEvent::BUNDLE_INSTALLED is fired.

4. The Bundle object for the newly or previously installed bundle is returned.

Remark An install location is an absolute path to a shared library or executable file which may contain several bundles, i.e. acts as a bundle library.

Remark If the bundleManifest is passed in, it is installed. In the event that the injected bundle manifest does NOT match the manifest in the bundle’s file, the behavior of the system is undefined.

Remark If the provided bundleManifest does not match the manifest embedded in the bundle’s file the behavior of that bundle in CppMicroServices is undefined.

Remark Example JSON representation of manifest AnyMap:

Return The Bundle objects of the installed bundle library.

Parameters

- location: The location of the bundle library to install.
- bundleManifest: OPTIONAL - the manifest of the bundle at “location”. If non-empty this will be used without opening the bundle at “location”. Otherwise, the bundle will be opened and the manifest read from there.

Exceptions

- std::runtime_error: If the BundleContext is no longer valid, or if the installation failed.
- std::logic_error: If the framework instance is no longer active
- std::invalid_argument: If the location is not a valid UTF8 string

15.1.4 BundleEvent

std::ostream &cppmicroservices::operator<<(std::ostream &os, BundleEvent::Type eventType)

Writes a string representation of eventType to the stream os.

std::ostream &cppmicroservices::operator<<(std::ostream &os, const BundleEvent &event)

Writes a string representation of event to the stream os.

class cppmicroservices::BundleEvent

#include <cppmicroservices/BundleEvent.h> An event from the Micro Services framework describing a bundle lifecycle change.

BundleEvent objects are delivered to listeners connected via BundleContext::AddBundleListener() when a change occurs in a bundles’s lifecycle. A type code is used to identify the event type for future extendability.
See `BundleContext::AddBundleListener`
The bundle’s `BundleActivator stop` method is about to be called if the bundle has a bundle activator class.

See `Bundle::Stop()`

`BUNDLE_LAZY_ACTIVATION` = 0x00000200

The bundle will be lazily activated.

The bundle has a *lazy activation policy* and is waiting to be activated. It is now in the `BUNDLE_STARTING` state and has a valid `BundleContext`.

Note This identifier is reserved for future use and not supported yet.

**Public Functions**

`BundleEvent ()`

Creates an invalid instance.

`operator bool () const`

Can be used to check if this `BundleEvent` instance is valid, or if it has been constructed using the default constructor.

Return `true` if this event object is valid, `false` otherwise.

`BundleEvent (Type type, const Bundle &bundle)`

Creates a bundle event of the specified type.

Parameters

- `type`: The event type.
- `bundle`: The bundle which had a lifecycle change. This bundle is used as the origin of the event.

`BundleEvent (Type type, const Bundle &bundle, const Bundle &origin)`

Creates a bundle event of the specified type.

Parameters

- `type`: The event type.
- `bundle`: The bundle which had a lifecycle change.
- `origin`: The bundle which is the origin of the event. For the event type `BUNDLE_INSTALLED`, this is the bundle whose context was used to install the bundle. Otherwise it is the bundle itself.

`Bundle GetBundle () const`

Returns the bundle which had a lifecycle change.

Return The bundle that had a change occur in its lifecycle.

`Type GetType () const`

Returns the type of lifecycle event.

The type values are:

- `BUNDLE_INSTALLED`
- `BUNDLE_RESOLVED`
• **BUNDLE_LAZY_ACTIVATION**
• **BUNDLE_STARTING**
• **BUNDLE_STARTED**
• **BUNDLE_STOPPING**
• **BUNDLE_STOPPED**
• **BUNDLE_UNRESOLVED**
• **BUNDLE_UNINSTALLED**

**Return** The type of lifecycle event.

**Bundle GetOrigin() const**

Returns the bundle that was the origin of the event.

For the event type **BUNDLE_INSTALLED**, this is the bundle whose context was used to install the bundle. Otherwise it is the bundle itself.

**Return** The bundle that was the origin of the event.

bool operator==(const BundleEvent &evt) const

Compares two bundle events for equality.

**Return** true if both events originate from the same bundle, describe a life-cycle change for the same bundle, and are of the same type. false otherwise. Two invalid bundle events are considered to be equal.

**Parameters**

• evt: The bundle event to compare this event with.

### 15.1.5 BundleEventHook

**struct cppmicroservices::BundleEventHook**

Bundle Event Hook Service.

Bundles registering this service will be called during bundle lifecycle (installed, starting, started, stopping, stopped, uninstalled) operations.

**Remark** Implementations of this interface are required to be thread-safe.

**Public Functions**

virtual ~BundleEventHook ()

**virtual bool Event(const BundleEvent &event, ShrinkableVector<BundleContext> &contexts) = 0**

Bundle event hook method.

This method is called prior to bundle event delivery when a bundle is installed, starting, started, stopping, stopped, and uninstalled. This method can filter the bundles which receive the event.

This method is called one and only one time for each bundle event generated, this includes bundle events which are generated when there are no bundle listeners registered.
Parameters

- **event**: The bundle event to be delivered.
- **contexts**: A list of Bundle Contexts for bundles which have listeners to which the specified event will be delivered. The implementation of this method may remove bundle contexts from the list to prevent the event from being delivered to the associated bundles.

### 15.1.6 BundleFindHook

**struct cppmicroservices::BundleFindHook**

Bundle Context Hook Service.

Bundles registering this service will be called during bundle find (get bundles) operations.

**Remark** Implementations of this interface are required to be thread-safe.

**Public Functions**

```cpp
virtual ~BundleFindHook() {}
virtual void Find(const BundleContext &context, ShrinkableVector<Bundle> &bundles) = 0
```

Find hook method.

This method is called for bundle find operations using BundleContext::GetBundle(long) and BundleContext::GetBundles() methods. The find method can filter the result of the find operation.

**Note** A find operation using the BundleContext::GetBundle(const std::string&) method does not cause the find method to be called, neither does any call to the static methods of the BundleRegistry class.

**Parameters**

- **context**: The bundle context of the bundle performing the find operation.
- **bundles**: A list of Bundles to be returned as a result of the find operation. The implementation of this method may remove bundles from the list to prevent the bundles from being returned to the bundle performing the find operation.

### 15.1.7 BundleResource

```cpp
std::ostream &cppmicroservices::operator<<(std::ostream &os, const BundleResource &resource)
```

Streams the resource path into the stream os.

**class cppmicroservices::BundleResource**

#include <cppmicroservices/BundleResource.h> Represents a resource (text file, image, etc.) embedded in a CppMicroServices bundle.

A BundleResource object provides information about a resource (external file) which was embedded into this bundle’s shared library. BundleResource objects can be obtained be calling Bundle::GetResource or Bundle::FindResources.

Example code for retrieving a resource object and reading its contents:
// Get this bundle's Bundle object
auto bundle = GetBundleContext().GetBundle();

BundleResource resource = bundle.GetResource("config.properties");
if (resource.IsValid()) {
    // Create a BundleResourceStream object
    BundleResourceStream resourceStream(resource);

    // Read the contents line by line
    std::string line;
    while (std::getline(resourceStream, line)) {
        // Process the content
        std::cout << line << std::endl;
    }
} else {
    // Error handling
}

BundleResource objects have value semantics and copies are very inexpensive.

See also:

BundleResourceStream
The Resource System

Public Functions

BundleResource()
Creates in invalid BundleResource object.

See IsValid()

BundleResource(const BundleResource &resource)
Copy constructor.

Parameters

- resource: The object to be copied.

~BundleResource()

BundleResource &operator=(const BundleResource &resource)
Assignment operator.

Return A reference to this BundleResource instance.

Parameters

- resource: The BundleResource object which is assigned to this instance.

bool operator<(const BundleResource &resource) const
A less than operator using the full resource path as returned by GetResourcePath() to define the ordering.

Return true if this BundleResource object is less then resource, false otherwise.
Parameters

• resource: The object to which this BundleResource object is compared to.

bool operator==(const BundleResource &resource) const
Equality operator for BundleResource objects.

Return true if this BundleResource object is equal to resource, i.e. they are coming from the same bundle (shared or static) and have an equal resource path, false otherwise.

Parameters

• resource: The object for testing equality.

bool operator!=(const BundleResource &resource) const
Inequality operator for BundleResource objects.

Return The result of !(*this == resource).

Parameters

• resource: The object for testing inequality.

bool isValid() const
Tests this BundleResource object for validity.

Invalid BundleResource objects are created by the default constructor or can be returned by the Bundle class if the resource path is not found.

Return true if this BundleResource object is valid and can safely be used, false otherwise.

operator bool() const
Boolean conversion operator using IsValid().

std::string GetName() const
Returns the name of the resource, excluding the path.

Example:

BundleResource resource = bundle->GetResource("/data/archive.tar.gz");
std::string name = resource.GetName(); // name = "archive.tar.gz"

Return The resource name.

See GetPath(), GetResourcePath()

std::string GetPath() const
Returns the resource’s path, without the file name.

Example:

BundleResource resource = bundle->GetResource("/data/archive.tar.gz");
std::string path = resource.GetPath(); // path = "/data/

The path with always begin and end with a forward slash.

Return The resource path without the name.
See `GetResourcePath()`, `GetName()` and `IsDir()`

std::string `GetResourcePath()` const  
Returns the resource path including the file name.

**Return** The resource path including the file name.

See `GetPath()`, `GetName()` and `IsDir()`

std::string `GetBaseName()` const  
Returns the base name of the resource without the path.

Example:

```cpp
BundleResource resource = bundle->GetResource("/data/archive.tar.gz");
std::string base = resource.GetBaseName();  // base = "archive"
```

**Return** The resource base name.

See `GetName()`, `GetSuffix()`, `GetCompleteSuffix()` and `GetCompleteBaseName()`

std::string `GetCompleteBaseName()` const  
Returns the complete base name of the resource without the path.

Example:

```cpp
BundleResource resource = bundle->GetResource("/data/archive.tar.gz");
std::string base = resource.GetCompleteBaseName();  // base = "archive.tar"
```

**Return** The resource’s complete base name.

See `GetName()`, `GetSuffix()`, `GetCompleteSuffix()`, and `GetBaseName()`

std::string `GetSuffix()` const  
Returns the suffix of the resource.

The suffix consists of all characters in the resource name after (but not including) the last ‘.’.

Example:

```cpp
BundleResource resource = bundle->GetResource("/data/archive.tar.gz");
std::string suffix = resource.GetSuffix();  // suffix = "gz"
```

**Return** The resource name suffix.

See `GetName()`, `GetCompleteSuffix()`, `GetBaseName()` and `GetCompleteBaseName()`

std::string `GetCompleteSuffix()` const  
Returns the complete suffix of the resource.

The suffix consists of all characters in the resource name after (but not including) the first ‘.’.

Example:

```cpp
BundleResource resource = bundle->GetResource("/data/archive.tar.gz");
std::string suffix = resource.GetCompleteSuffix();  // suffix = "tar.gz"
```
Return  The resource name suffix.

See  GetName(), GetSuffix(), GetBaseName(), and GetCompleteBaseName()

bool IsDir() const
Returns true if this BundleResource object points to a directory and thus may have child resources.

Return  true if this object points to a directory, false otherwise.

bool IsFile() const
Returns true if this BundleResource object points to a file resource.

Return  true if this object points to an embedded file, false otherwise.

std::vector<std::string> GetChildren() const
Returns a list of resource names which are children of this object.

The returned names are relative to the path of this BundleResource object and may contain file as well as directory entries.

Return  A list of child resource names.

std::vector<BundleResource> GetChildResources() const
Returns a list of resource objects which are children of this object.

The returned BundleResource objects may contain files as well as directory resources.

Return  A list of child resource objects.

int GetSize() const
Returns the (uncompressed) size of the resource data for this BundleResource object.

Return  The uncompressed resource data size.

int GetCompressedSize() const
Returns the compressed size of the resource data for this BundleResource object.

Return  The compressed resource data size.

time_t GetLastModified() const
Returns the last modified time of this resource in seconds from the epoch.

Return  Last modified time of this resource.

uint32_t GetCrc32() const
Returns the CRC-32 checksum of this resource.

Return  CRC-32 checksum of this resource.


Friends

friend gr_bundleresource::std::hash< BundleResource >

template<>
struct std::hash<cppmicroservices::BundleResource>

#include <cppmicroservices/BundleResource.h> Hash functor specialization for BundleResource objects.

15.1.8 BundleResourceStream

class cppmicroservices::BundleResourceStream
An input stream class for BundleResource objects.

This class provides access to the resource data embedded in a bundle's shared library via a STL input stream interface.

See BundleResource for an example how to use this class.

Inherits from cppmicroservices::detail::BundleResourceBuffer, std::istream

Public Functions


Parameters

• resource: The BundleResource object for which an input stream should be constructed.
• mode: The open mode of the stream. If std::ios_base::binary is used, the resource data will be treated as binary data, otherwise the data is interpreted as text data and the usual platform specific end-of-line translations take place.

15.1.9 Framework

class cppmicroservices::Framework
A Framework instance.

A Framework is itself a bundle and is known as the “System Bundle”. The System Bundle differs from other bundles in the following ways:

• The system bundle is always assigned a bundle identifier of zero (0).
• The system bundle getLocation method returns the string: “System Bundle”.
• The system bundle’s life cycle cannot be managed like normal bundles. Its life cycle methods behave as follows:
  – Start - Initialize the framework and start installed bundles.
  – Stop - Stops all installed bundles.
- Uninstall - The Framework throws a std::runtime_error exception indicating that the system bundle cannot be uninstalled.

Framework instances are created using a FrameworkFactory. The methods of this class can be used to manage and control the created framework instance.

**Remark** This class is thread-safe.

**See** FrameworkFactory::NewFramework(const std::map<std::string, Any>& configuration)

Inherits from cppmicroservices::Bundle

## Public Functions

**Framework (Bundle b)**

Convert a Bundle representing the system bundle to a Framework instance.

**Parameters**

- b: The system bundle

**Exceptions**

- std::logic_error: If the bundle is not the system bundle.

**Framework (const Framework &fw)**

**Framework (Framework &&fw)**

**Framework &operator=(const Framework &fw)**

**Framework &operator=(Framework &&fw)**

void **Init ()**

Initialize this Framework.

After calling this method, this Framework has:

- Generated a new framework UUID.
- Moved to the STATE_STARTING state.
- A valid Bundle Context.
- Event handling enabled.
- Reified Bundle objects for all installed bundles.
- Registered any framework services.

This Framework will not actually be started until Start is called.

This method does nothing if called when this Framework is in the STATE_STARTING, STATE_ACTIVE or STATE_STOPPING states.

**Exceptions**

- std::runtime_error: If this Framework could not be initialized.
FrameworkEvent WaitForStop(const std::chrono::milliseconds &timeout)

Wait until this Framework has completely stopped.

The Stop method on a Framework performs an asynchronous stop of the Framework if it was built with threading support.

This method can be used to wait until the asynchronous stop of this Framework has completed. This method will only wait if called when this Framework is in the STATE_STARTING, STATE_ACTIVE, or STATE_STOPPING states. Otherwise it will return immediately.

A Framework Event is returned to indicate why this Framework has stopped.

Return A Framework Event indicating the reason this method returned. The following FrameworkEvent types may be returned by this method.

- FRAMEWORK_STOPPED - This Framework has been stopped.
- FRAMEWORK_ERROR - The Framework encountered an error while shutting down or an error has occurred which forced the framework to shutdown.
- FRAMEWORK_WAIT_TIMEDOUT - This method has timed out and returned before this Framework has stopped.

Parameters

- timeout: Maximum time duration to wait until this Framework has completely stopped. A value of zero will wait indefinitely.

15.1.10 FrameworkEvent

std::ostream &cppmicroservices::operator<<(std::ostream &os, FrameworkEvent::Type eventType)

Writes a string representation of eventType to the stream os.

std::ostream &cppmicroservices::operator<<(std::ostream &os, const FrameworkEvent &evt)

Writes a string representation of evt to the stream os.

bool cppmicroservices::operator==(const FrameworkEvent &rhs, const FrameworkEvent &lhs)

Compares two framework events for equality.

class cppmicroservices::FrameworkEvent

#include <cppmicroservices/FrameworkEvent.h> An event from the Micro Services framework describing a Framework event.

FrameworkEvent objects are delivered to listeners connected via BundleContext::AddFrameworkListener() when an event occurs within the Framework which a user would be interested in. A Type code is used to identify the event type for future extendability.

See BundleContext::AddFrameworkListener

Public Types

enum Type

A type code used to identify the event type for future extendability.

Values:
FRAMEWORK_STARTED = 0x00000001
The Framework has started.
This event is fired when the Framework has started after all installed bundles that are marked to be started have been started. The source of this event is the System Bundle.

FRAMEWORK_ERROR = 0x00000002
An error has occurred.
There was an error associated with a bundle.

FRAMEWORK_WARNING = 0x00000010
A warning has occurred.
There was a warning associated with a bundle.

FRAMEWORK_INFO = 0x00000020
An informational event has occurred.
There was an informational event associated with a bundle.

FRAMEWORK_STOPPED = 0x00000040
The Framework has been stopped.
This event is fired when the Framework has been stopped because of a stop operation on the system bundle. The source of this event is the System Bundle.

FRAMEWORK_STOPPED_UPDATE = 0x00000080
The Framework is about to be stopped.
This event is fired when the Framework has been stopped because of an update operation on the system bundle. The Framework will be restarted after this event is fired. The source of this event is the System Bundle.

FRAMEWORK_WAIT_TIMEDOUT = 0x00000200
The Framework did not stop before the wait timeout expired.
This event is fired when the Framework did not stop before the wait timeout expired. The source of this event is the System Bundle.

Public Functions

FrameworkEvent()
Creates an invalid instance.

operator bool() const
Returns false if the FrameworkEvent is empty (i.e invalid) and true if the FrameworkEvent is not null and contains valid data.

Return true if this event object is valid, false otherwise.

FrameworkEvent(Type type, const Bundle &bundle, const std::string &message, const std::exception_ptr exception = nullptr)
Creates a Framework event of the specified type.

Parameters
• type: The event type.
• bundle: The bundle associated with the event. This bundle is also the source of the event.
- message: The message associated with the event.
- exception: The exception associated with this event. Should be nullptr if there is no exception.

Bundle GetBundle() const

Returns the bundle associated with the event.

Return The bundle associated with the event.

std::string GetMessage() const

Returns the message associated with the event.

Return the message associated with the event.

std::exception_ptr GetThrowable() const

Returns the exception associated with this event.

Remark Use std::rethrow_exception to throw the exception returned.

Return The exception. May be nullptr if there is no related exception.

Type GetType() const

Returns the type of framework event.

The type values are:

- FRAMEWORK_STARTED
- FRAMEWORK_ERROR
- FRAMEWORK_WARNING
- FRAMEWORK_INFO
- FRAMEWORK_STOPPED
- FRAMEWORK_STOPPED_UPDATE
- FRAMEWORK_WAIT_TIMEDOUT

Return The type of Framework event.

15.1.11 PrototypeServiceFactory

struct cppmicroservices::PrototypeServiceFactory

A factory for prototype scope services.

The factory can provide multiple, unique service objects.

When registering a service, a PrototypeServiceFactory object can be used instead of a service object, so that the bundle developer can create a unique service object for each caller that is using the service. When a caller uses a ServiceObjects to request a service instance, the framework calls the GetService method to return a service object specifically for the requesting caller. The caller can release the returned service object and the framework will call the UngetService method with the service object. When a bundle uses the GetService(BundleContext::GetService(const ServiceReferenceBase&)) method to obtain a service object, the framework acts as if the service has bundle scope. That is, the framework will call the GetService method to obtain a bundle-scoped instance which will be cached and have a use count. See ServiceFactory.
A bundle can use both `ServiceObjects` and `BundleContext::GetService(const ServiceReferenceBase&)` to obtain a service object for a service. `ServiceObjects::GetService()` will always return an instance provided by a call to `GetService(const Bundle&, const ServiceRegistrationBase&)` and `BundleContext::GetService(const ServiceReferenceBase&)` will always return the bundle-scoped instance. `PrototypeServiceFactory` objects are only used by the framework and are not made available to other bundles. The framework may concurrently call a `PrototypeServiceFactory`.

See `BundleContext::GetServiceObjects()`
See `ServiceObjects`

Inherits from `cppmicroservices::ServiceFactory`

**Public Functions**

```cpp
InterfaceMapConstPtr getService(const Bundle &bundle, const ServiceRegistrationBase &registration) = 0
```

Returns a service object for a caller.

The framework invokes this method for each caller requesting a service object using `ServiceObjects::GetService()`. The factory can then return a specific service object for the caller. The framework checks that the returned service object is valid. If the returned service object is empty or does not contain entries for all the interfaces named when the service was registered, a warning is issued and `nullptr` is returned to the caller. If this method throws an exception, a warning is issued and `nullptr` is returned to the caller.

**Return** A service object that must contain entries for all the interfaces named when the service was registered.

See `ServiceObjects::GetService()`
See `InterfaceMapConstPtr`

**Parameters**

- `bundle`: The bundle requesting the service.
- `registration`: The `ServiceRegistrationBase` object for the requested service.

```cpp
void UngetService(const Bundle &bundle, const ServiceRegistrationBase &registration, const InterfaceMapConstPtr &service) = 0
```

Releases a service object created for a caller.

The framework invokes this method when a service has been released by a bundles such as by calling `ServiceObjects::UngetService()`. The service object may then be destroyed. If this method throws an exception, a warning is issued.

See `ServiceObjects::UngetService()`

**Parameters**

- `bundle`: The bundle releasing the service.
- `registration`: The `ServiceRegistrationBase` object for the service being released.
- `service`: The service object returned by a previous call to the `GetService` method.
15.1.12 ServiceEvent

std::ostream &cppmicroservices::operator<<(std::ostream &os, const ServiceEvent::Type &type)
Writes a string representation of type to the stream os.

std::ostream &cppmicroservices::operator<<(std::ostream &os, const ServiceEvent &event)
Writes a string representation of event to the stream os.

class cppmicroservices::ServiceEvent
#include <cppmicroservices/ServiceEvent.h> An event from the Micro Services framework describing a service lifecycle change.

ServiceEvent objects are delivered to listeners connected via BundleContext::AddServiceListener() when a change occurs in this service’s lifecycle. A type code is used to identify the event type for future extendability.

Public Types

enum Type
The service event type.
Values:

SERVICE_REGISTERED = 0x00000001
This service has been registered.

This event is delivered after the service has been registered with the framework.

See BundleContext::RegisterService

SERVICE_MODIFIED = 0x00000002
The properties of a registered service have been modified.

This event is delivered after the service properties have been modified.

See ServiceRegistration::SetProperties

SERVICE_UNREGISTERING = 0x00000004
This service is in the process of being unregistered.

This event is delivered before the service has completed unregistering.

If a bundle is using a service that is SERVICE_UNREGISTERING, the bundle should release its use of the service when it receives this event. If the bundle does not release its use of the service when it receives this event, the framework will automatically release the bundle’s use of the service while completing the service unregistration operation.

See ServiceRegistration::Unregister

SERVICE_MODIFIED_ENDMATCH = 0x00000008
The properties of a registered service have been modified and the new properties no longer match the listener’s filter.

This event is delivered after the service properties have been modified. This event is only delivered to listeners which were added with a non-empty filter where the filter matched the service properties prior to the modification but the filter does not match the modified service properties.

See ServiceRegistration::SetProperties
Public Functions

ServiceEvent ()
Creates an invalid instance.

operator bool () const
Can be used to check if this ServiceEvent instance is valid, or if it has been constructed using the default constructor.

Returns true if this event object is valid, false otherwise.

ServiceEvent (Type type, const ServiceReferenceBase &reference)
Creates a new service event object.

Parameters

• type: The event type.
• reference: A ServiceReference object to the service that had a lifecycle change.

ServiceEvent (const ServiceEvent &other)

ServiceEvent &operator=(const ServiceEvent &other)

ServiceReferenceU GetServiceReference () const
Returns a reference to the service that had a change occur in its lifecycle.

This reference is the source of the event.

Returns Reference to the service that had a lifecycle change.

template <class S>
ServiceReference<S> GetServiceReference () const

Type GetType () const
Returns the type of event.

The event type values are:

• SERVICE_REGISTERED
• SERVICE_MODIFIED
• SERVICE_MODIFIED_ENDMATCH
• SERVICE_UNREGISTERING

Returns Type of service lifecycle change.

15.1.13 ServiceEventListenerHook

struct cppmicroservices::ServiceEventListenerHook
Service Event Listener Hook Service.

Bundles registering this service will be called during service (register, modify, and unregister service) operations.

Remark Implementations of this interface are required to be thread-safe.
Public Types

using ShrinkableMapType = ShrinkableMap.BundleContext, ShrinkableVector.ServiceListenerHook::ListenerInfo>>

ShrinkableMap type for filtering event listeners.

Public Functions

virtual ~ServiceEventListenerHook() = default

virtual void Event (const ServiceEvent &event, ShrinkableMapType &listeners) = 0

Event listener hook method.

This method is called prior to service event delivery when a publishing bundle registers, modifies or un-registers a service. This method can filter the listeners which receive the event.

Parameters

• event: The service event to be delivered.

• listeners: A map of Bundle Contexts to a list of Listener Infos for the bundle’s listeners to which the specified event will be delivered. The implementation of this method may remove bundle contexts from the map and listener infos from the list values to prevent the event from being delivered to the associated listeners.

15.1.14 ServiceException

std::ostream &operator<< (std::ostream &os, const cppmicroservices::ServiceException &exc)

Writes a string representation of exc to the stream os.

class cppmicroservices::ServiceException

#include <cppmicroservices/ServiceException.h> A service exception used to indicate that a service problem occurred.

A ServiceException object is created by the framework or to denote an exception condition in the service. An enum type is used to identify the exception type for future extendability.

This exception conforms to the general purpose exception chaining mechanism.

Inherits from std::runtime_error

Public Types

enum Type

Values:

UNSPECIFIED = 0

No exception type is unspecified.

UNREGISTERED = 1

The service has been unregistered.

FACTORY_ERROR = 2

The service factory produced an invalid service object.

FACTORY_EXCEPTION = 3

The service factory threw an exception.
REMOTE = 5
An error occurred invoking a remote service.

FACTORY_RECURSION = 6
The service factory resulted in a recursive call to itself for the requesting bundle.

Public Functions

ServiceException (const std::string &msg, const Type &type = UNSPECIFIED)
Creates a ServiceException with the specified message, type and exception cause.

Parameters

• msg: The associated message.
• type: The type for this exception.

ServiceException (const ServiceException &o)

ServiceException &operator= (const ServiceException &o)

~ServiceException ()

Type GetType () const
Returns the type for this exception or UNSPECIFIED if the type was unspecified or unknown.

Return The type of this exception.

15.1.15 ServiceFactory

class cppmicroservices::ServiceFactory
A factory for bundle scope services.

The factory can provide service objects unique to each bundle.

When registering a service, a ServiceFactory object can be used instead of a service object, so that the bundle developer can create a customized service object for each bundle that is using the service.

When a bundle requests the service object, the framework calls the ServiceFactory::GetService method to return a service object customized for the requesting bundle. The returned service object is cached by the framework for subsequent calls to BundleContext::GetService(const ServiceReference&) until the bundle releases its use of the service.

When the bundle’s use count for the service is decremented to zero (including the bundle stopping or the service being unregistered), the framework will call the ServiceFactory::UngetService method.

ServiceFactory objects are only used by the framework and are not made available to other bundles in the bundle environment. The framework may concurrently call a ServiceFactory.

See BundleContext::GetService
See PrototypeServiceFactory
Remark This class is thread safe.

Subclassed by cppmicroservices::PrototypeServiceFactory
Public Functions

virtual ~ServiceFactory()

virtual InterfaceMapConstPtr GetService(const Bundle &bundle, const ServiceRegistrationBase &registration) = 0

Returns a service object for a bundle.

The framework invokes this method the first time the specified bundle requests a service object using the BundleContext::GetService(const ServiceReferenceBase&) method. The factory can then return a customized service object for each bundle.

The framework checks that the returned service object is valid. If the returned service object is null or does not contain entries for all the classes named when the service was registered, a framework event of type FrameworkEvent::FRAMEWORK_ERROR is fired containing a service exception of type ServiceException::FACTORY_ERROR and null is returned to the bundle. If this method throws an exception, a framework event of type FrameworkEvent::FRAMEWORK_ERROR is fired containing a service exception of type ServiceException::FACTORY_EXCEPTION with the thrown exception as a nested exception and null is returned to the bundle. If this method is recursively called for the specified bundle, a framework event of type FrameworkEvent::FRAMEWORK_ERROR is fired containing a service exception of type ServiceException::FACTORY_RECURSION and null is returned to the bundle.

The framework caches the valid service object, and will return the same service object on any future call to BundleContext::GetService for the specified bundle. This means the framework does not allow this method to be concurrently called for the specified bundle.

Return A service object that must contain entries for all the interfaces named when the service was registered.

See BundleContext::GetService

See InterfaceMapConstPtr

Parameters

• bundle: The bundle requesting the service.

• registration: The ServiceRegistrationBase object for the requested service.

virtual void UngetService(const Bundle &bundle, const ServiceRegistrationBase &registration, const InterfaceMapConstPtr &service) = 0

Releases a service object customized for a bundle.

The Framework invokes this method when a service has been released by a bundle. If this method throws an exception, a framework event of type FrameworkEvent::FRAMEWORK_ERROR is fired containing a service exception of type ServiceException::FACTORY_EXCEPTION with the thrown exception as a nested exception.

See InterfaceMapConstPtr

Parameters

• bundle: The Bundle releasing the service.

• registration: The ServiceRegistration object for the service being released.

• service: The service object returned by a previous call to the ServiceFactory::GetService method.
15.1.16 ServiceFindHook

struct cppmicroservices::ServiceFindHook
Service Find Hook Service.

Bundles registering this service will be called during service find (get service references) operations.

Remark Implementations of this interface are required to be thread-safe.

Public Functions

virtual ~ServiceFindHook() = 0
virtual void Find(const BundleContext &context, const std::string &name, const std::string &filter,
ShrinkableVector<ServiceReferenceBase> &references) = 0

Find hook method.
This method is called during the service find operation (for example, BundleContext::GetServiceReferences<S>()). This method can filter the result of the find operation.

Parameters

• context: The bundle context of the bundle performing the find operation.
• name: The class name of the services to find or an empty string to find all services.
• filter: The filter criteria of the services to find or an empty string for no filter criteria.
• references: A list of Service References to be returned as a result of the find operation.

The implementation of this method may remove service references from the list to prevent the references from being returned to the bundle performing the find operation.

15.1.17 ServiceInterface

template <class T>
std::shared_ptr<ServiceFactory> cppmicroservices::ToFactory(const std::shared_ptr<T> &factory)

Cast the argument to a shared pointer of type ServiceFactory.
Useful when calling BundleContext::RegisterService with a service factory, for example:

std::shared_ptr<MyServiceFactory> factory = std::make_shared<MyServiceFactory>();
context->RegisterService<ISomeInterface>(ToFactory(factory));

Return A shared_ptr object of type ServiceFactory

See BundleContext::RegisterService(ServiceFactory* factory, const ServiceProperties& properties)

Parameters

• factory: The service factory shared_ptr object

using cppmicroservices::InterfaceMap = typedef std::unordered_map<std::string, std::shared_ptr<void> >
A map containing interfaces ids and their corresponding service object smart pointers.

InterfaceMap instances represent a complete service object which implements one or more service interfaces.
For each implemented service interface, there is an entry in the map with the key being the service interface id and the value a smart pointer to the service interface implementation.
To create InterfaceMap instances, use the `MakeInterfaceMap` helper class.

**Note** This is a low-level type and should only rarely be used.

See `MakeInterfaceMap`

```cpp
template <class T>
const std::string &
us_service_interface_iid()

Returns a unique id for a given type.

By default, the demangled name of `T` is returned.

This template method may be specialized directly or by using the macro `CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE` to return a custom id for each service interface.

**Return** A unique id for the service interface type `T`.

**Template Parameters**

- `T`: The service interface type.

```cpp
template <class Interface>
std::shared_ptr<Interface> cppmicroservices::ExtractInterface(const InterfaceMapConstPtr &map)

Extract a service interface pointer from a given InterfaceMap instance.

**Return** A shared pointer object of type `Interface`. The returned object is empty if the map does not contain an entry for the given type.

See `MakeInterfaceMap`

**Parameters**

- `map`: a InterfaceMap instance.

```cpp
std::shared_ptr<void> cppmicroservices::ExtractInterface(const InterfaceMapConstPtr &map, const std::string &interfaceId)

Extract a service interface pointer from a given InterfaceMap instance.

**Return** The service interface pointer for the service interface id or `nullptr` if map does not contain an entry for the given type.

See `ExtractInterface(const InterfaceMapConstPtr&)`

**Parameters**

- `map`: a InterfaceMap instance.
- `interfaceId`: The interface id string.

```cpp
template <class... Interfaces>
class cppmicroservices::MakeInterfaceMap
#include <cppmicroservices/ServiceInterface.h> Helper class for constructing InterfaceMap instances based on service implementations or service factories.

Example usage:

```cpp
std::shared_ptr<MyService> service; // implements I1 and I2
InterfaceMap im = MakeInterfaceMap<I1,I2>(service);
```

See `InterfaceMap`
Public Functions

```cpp
template <class Impl>
MakeInterfaceMap (const std::shared_ptr<Impl> & impl)
Constructor taking a service implementation pointer.
```

Parameters

- `impl`: A service implementation pointer, which must be castable to a all specified service interfaces.

```cpp
MakeInterfaceMap (std::shared_ptr<ServiceFactory> factory)
Constructor taking a service factory.
```

Parameters

- `factory`: A service factory.

```cpp
operator InterfaceMapPtr()
operator InterfaceMapConstPtr()
```

```cpp
CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE(_service_interface_type, _service_interface_id)
Declare a service interface id.
```

This macro associates the given identifier `_service_interface_id` (a string literal) to the interface class called `_service_interface_type`. The Identifier must be unique. For example:

```cpp
#include "cppmicroservices/ServiceInterface.h"

struct ISomeInterface { ... };

CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE(ISomeInterface, "com.mycompany.service.ISomeInterface/1.0")
```

The usage of this macro is optional and the service interface id which is automatically associated with any type is usually good enough (the demangled type name). However, care must be taken if the default id is compared with a string literal hard-coding a service interface id. E.g. the default id for templated types in the STL may differ between platforms. For user-defined types and templates the ids are typically consistent, but platform specific default template arguments will lead to different ids.

This macro is normally used right after the class definition for `_service_interface_type`, in a header file.

If you want to use `CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE` with interface classes declared in a namespace then you have to make sure the `CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE` macro call is not inside a namespace though. For example:

```cpp
#include "cppmicroservices/ServiceInterface.h"

namespace Foo
{
  struct ISomeInterface { ... };
}

CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE(Foo::ISomeInterface, "com.mycompany.service.ISomeInterface/1.0")
```
Parameters

- `_service_interface_type`: The service interface type.
- `_service_interface_id`: A string literal representing a globally unique identifier.

### 15.1.18 ServiceListenerHook

```cpp
struct cppmicroservices::ServiceListenerHook
#include <cppmicroservices/ServiceListenerHook.h> Service Listener Hook Service.

 Bundles registering this service will be called during service listener addition and removal.

**Remark** Implementations of this interface are required to be thread-safe.

#### Public Functions

```cpp
virtual ~ServiceListenerHook() = default
```

```cpp
virtual void Added (const std::vector<ListenerInfo> &listeners) = 0
```

Added listeners hook method.

This method is called to provide the hook implementation with information on newly added service listeners. This method will be called as service listeners are added while this hook is registered. Also, immediately after registration of this hook, this method will be called to provide the current collection of service listeners which had been added prior to the hook being registered.

**Parameters**

- `listeners`: A collection of `ListenerInfo` objects for newly added service listeners which are now listening to service events.

```cpp
virtual void Removed (const std::vector<ListenerInfo> &listeners) = 0
```

Removed listeners hook method.

This method is called to provide the hook implementation with information on newly removed service listeners. This method will be called as service listeners are removed while this hook is registered.

**Parameters**

- `listeners`: A collection of `ListenerInfo` objects for newly removed service listeners which are no longer listening to service events.

```cpp
struct ListenerInfo
#include <cppmicroservices/ServiceListenerHook.h> Information about a Service Listener.

 This class describes the bundle which added the Service Listener and the filter with which it was added.

**Remark** This class is not intended to be implemented by clients.
Public Functions

**ListenerInfo()**

**ListenerInfo(const ListenerInfo &other)**

**~ListenerInfo()**

**ListenerInfo &operator=(const ListenerInfo &other)**

bool **IsNull() const**

Can be used to check if this ListenerInfo instance is valid, or if it has been constructed using the default constructor.

**Return** true if this listener object is valid, false otherwise.

**BundleContext GetBundleContext() const**

Return the context of the bundle which added the listener.

**Return** The context of the bundle which added the listener.

std::string **GetFilter() const**

Return the filter string with which the listener was added.

**Return** The filter string with which the listener was added. This may be empty if the listener was added without a filter.

bool **IsRemoved() const**

Return the state of the listener for this addition and removal life cycle.

Initially this method will return false indicating the listener has been added but has not been removed. After the listener has been removed, this method must always returns true.

There is an extremely rare case in which removed notification to ServiceListenerHooks can be made before added notification if two threads are racing to add and remove the same service listener. Because ServiceListenerHooks are called synchronously during service listener addition and removal, the CppMicroServices library cannot guarantee in-order delivery of added and removed notification for a given service listener. This method can be used to detect this rare occurrence.

**Return** false if the listener has not been removed, true otherwise.

bool **operator==(const ListenerInfo &other) const**

Compares this ListenerInfo to another ListenerInfo.

Two ListenerInfos are equal if they refer to the same listener for a given addition and removal life cycle. If the same listener is added again, it will have a different ListenerInfo which is not equal to this ListenerInfo.

**Return** true if the other object is a ListenerInfo object and both objects refer to the same listener for a given addition and removal life cycle.

**Parameters**

- other: The object to compare against this ListenerInfo.

Friends

friend gr_servicelistenerhook::std::hash< ServiceListenerHook::ListenerInfo > template<>
struct std::hash<cppmicroservices::ServiceListenerHook::ListenerInfo>
#include <cppmicroservices/ServiceListenerHook.h> Hash functor specialization for ServiceListenerHook::ListenerInfo objects.

15.1.19 ServiceObjects

template <class S>
class cppmicroservices::ServiceObjects
#include <cppmicroservices/ServiceObjects.h> Allows multiple service objects for a service to be obtained.

For services with prototype scope, multiple service objects for the service can be obtained. For services with singleton or bundle scope, only one, use-counted service object is available. Any unreleased service objects obtained from this ServiceObjects object are automatically released by the framework when the bundles associated with the BundleContext used to create this ServiceObjects object is stopped.

Template Parameters
- S: Type of Service.

Inherits from cppmicroservices::ServiceObjectsBase

Public Functions

ServiceObjects (const ServiceObjects &other)
ServiceObjects &operator= (const ServiceObjects &other)
ServiceObjects (ServiceObjects &&other)
ServiceObjects &operator= (ServiceObjects &&other)

std::shared_ptr<S> GetService () const
Returns a service object for the referenced service.

This ServiceObjects object can be used to obtain multiple service objects for the referenced service if the service has prototype scope. If the referenced service has singleton or bundle scope, this method behaves the same as calling the BundleContext::GetService(const ServiceReferenceBase&) method for the referenced service. That is, only one, use-counted service object is available from this ServiceObjects object.

This method will always return nullptr when the referenced service has been unregistered.

For a prototype scope service, the following steps are taken to get the service object:

1. If the referenced service has been unregistered, nullptr is returned.
2. The PrototypeServiceFactory::GetService(const Bundle&, const ServiceRegistrationBase&) method is called to create a service object for the caller.
3. If the service object (an instance of InterfaceMap) returned by the PrototypeServiceFactory object is empty, does not contain all the interfaces named when the service was registered or the PrototypeServiceFactory object throws an exception, nullptr is returned and a warning message is issued.
4. The service object is returned.

Return A shared_ptr to the service object. The returned shared_ptr is empty if the service is not registered, the service object returned by a ServiceFactory does not contain all the classes under which it was registered or the ServiceFactory threw an exception.
Exceptions

- std::logic_error: If the BundleContext used to create this ServiceObjects object is no longer valid.

ServiceReference<S> GetServiceReference() const
Returns the ServiceReference for this ServiceObjects object.

Return The ServiceReference for this ServiceObjects object.

template <>
template<>
class cppmicroservices::ServiceObjects<void>
#include <cppmicroservices/ServiceObjects.h> Allows multiple service objects for a service to be obtained.
This is a specialization of the ServiceObjects class template for void, which maps to all service interface types.

See ServiceObjects

Inherits from cppmicroservices::ServiceObjectsBase

Public Functions

ServiceObjects(const ServiceObjects &other)
ServiceObjects &operator=(const ServiceObjects &other)
ServiceObjects(ServiceObjects &&other)
ServiceObjects &operator=(ServiceObjects &&other)

InterfaceMapConstPtr GetService() const
Returns a service object as a InterfaceMap instance for the referenced service.
This method is the same as ServiceObjects<S>::GetService() except for the return type. Further, this method will always return an empty InterfaceMap object when the referenced service has been unregistered.

Return A InterfaceMapConstPtr object for the referenced service, which is empty if the service is not registered, the InterfaceMap returned by a ServiceFactory does not contain all the classes under which the service object was registered or the ServiceFactory threw an exception.

See ServiceObjects<S>::GetService()

Exceptions

- std::logic_error: If the BundleContext used to create this ServiceObjects object is no longer valid.

ServiceReferenceU GetServiceReference() const
Returns the ServiceReference for this ServiceObjects object.

Return The ServiceReference for this ServiceObjects object.
15.1.20 ServiceReference

typedef ServiceReference<void> cppmicroservices::ServiceReferenceU
A service reference of unknown type, which is not bound to any interface identifier.

std::ostream &cppmicroservices::operator<<(std::ostream &os, const ServiceReference &serviceRef)
Writes a string representation of serviceRef to the stream os.

template <class S>
class cppmicroservices::ServiceReference
#include <cppmicroservices/ServiceReference.h> A reference to a service.

The framework returns ServiceReference objects from the BundleContext::GetServiceReference
and BundleContext::GetServiceReferences methods.

A ServiceReference object may be shared between bundles and can be used to examine the properties of
the service and to get the service object.

Every service registered in the framework has a unique ServiceRegistration object and may have mul-
tiple, distinct ServiceReference objects referring to it. ServiceReference objects associated with a
ServiceRegistration are considered equal (more specifically, their operator==() method will return
true when compared).

If the same service object is registered multiple times, ServiceReference objects are not equal.

See BundleContext::GetServiceReference
See BundleContext::GetServiceReferences
See BundleContext::GetService

Template Parameters
• S: The class type of the service interface

Inherits from cppmicroservices::ServiceReferenceBase

Public Types

template<>
using ServiceType = S

Public Functions

ServiceReference()
Creates an invalid ServiceReference object.

You can use this object in boolean expressions and it will evaluate to false.

ServiceReference(const ServiceReference&)  
ServiceReference &operator=(const ServiceReference&)  
ServiceReference(const ServiceReferenceBase &base)
class cppmicroservices::ServiceReferenceBase
#include <cppmicroservices/ServiceReferenceBase.h> A reference to a service.

Note: This class is provided as public API for low-level service queries only. In almost all cases you should use the template ServiceReference instead.

Subclassed by cppmicroservices::ServiceReference< S >, cppmicroservices::ServiceReference< cppmicroservices::service::cm::ConfigurationAdmin >

Public Functions

ServiceReferenceBase (const ServiceReferenceBase &ref)

operator bool () const
Converts this ServiceReferenceBase instance into a boolean expression.
If this instance was default constructed or the service it references has been unregistered, the conversion returns false, otherwise it returns true.

ServiceReferenceBase &operator= (std::nullptr_t)
Releases any resources held or locked by this ServiceReferenceBase and renders it invalid.

~ServiceReferenceBase ()

Any GetProperty (const std::string &key) const
Returns the property value to which the specified property key is mapped in the properties ServiceProperties object of the service referenced by this ServiceReferenceBase object.
Property keys are case-insensitive.
This method continues to return property values after the service has been unregistered. This is so references to unregistered services can still be interrogated.

Return The property value to which the key is mapped; an invalid Any if there is no property named after the key.

Parameters
• key: The property key.

void GetPropertyKeys (std::vector<std::string> &keys) const
Returns a list of the keys in the ServiceProperties object of the service referenced by this ServiceReferenceBase object.
This method will continue to return the keys after the service has been unregistered. This is so references to unregistered services can still be interrogated.

Parameters
• keys: A vector being filled with the property keys.

std::vector<std::string> GetPropertyKeys () const
Returns a list of the keys in the ServiceProperties object of the service referenced by this ServiceReferenceBase object.
This method will continue to return the keys after the service has been unregistered. This is so references to unregistered services can still be interrogated.
Return A vector being filled with the property keys.

Bundle GetBundle() const
Returns the bundle that registered the service referenced by this ServiceReferenceBase object.
This method must return an invalid bundle when the service has been unregistered. This can be used to
determine if the service has been unregistered.

Return The bundle that registered the service referenced by this ServiceReferenceBase object; an
invalid bundle if that service has already been unregistered.

See BundleContext::RegisterService(const InterfaceMap&, const ServiceProperties&)
See Bundle::operator bool() const

std::vector<Bundle> GetUsingBundles() const
Returns the bundles that are using the service referenced by this ServiceReferenceBase object.
Specifically, this method returns the bundles whose usage count for that service is greater than zero.

Return A list of bundles whose usage count for the service referenced by this
ServiceReferenceBase object is greater than zero.

std::string GetInterfaceId() const
Returns the interface identifier this ServiceReferenceBase object is bound to.
A default constructed ServiceReferenceBase object is not bound to any interface identifier and calling this
method will return an empty string.

Return The interface identifier for this ServiceReferenceBase object.

bool IsConvertibleTo(const std::string &interfaceid) const
Checks whether this ServiceReferenceBase object can be converted to another ServiceReferenceBase ob-
ject, which will be bound to the given interface identifier.

ServiceReferenceBase objects can be converted if the underlying service implementation was registered
under multiple service interfaces.

Return true if this ServiceReferenceBase object can be converted, false otherwise.

Parameters
• interfaceid:

bool operator< (const ServiceReferenceBase &reference) const
Compares this ServiceReferenceBase with the specified ServiceReferenceBase for order.
If this ServiceReferenceBase and the specified ServiceReferenceBase have the same service id they are equal. This ServiceReferenceBase is less than the specified ServiceReferenceBase if it has a lower service ranking and greater if it has a higher service ranking. Otherwise, if this ServiceReferenceBase and the specified ServiceReferenceBase have the same service ranking, this ServiceReferenceBase is less than the specified ServiceReferenceBase if it has a higher service id and greater if it has a lower service id.

Return Returns a false or true if this ServiceReferenceBase is less than or greater than the specified
ServiceReferenceBase.

Parameters
• reference: The ServiceReferenceBase to be compared.

bool operator==(const ServiceReferenceBase &reference) const

ServiceReferenceBase &operator=(const ServiceReferenceBase &reference)

Friends

friend gr_servicereference::::std::hash< ServiceReferenceBase >

template<> struct std::hash<cppmicroservices::ServiceReferenceBase>
    #include <cppmicroservices/ServiceReferenceBase.h> Hash functor specialization for ServiceReferenceBase objects.

15.1.21 ServiceRegistration

template <class I1, class... Interfaces>
class cppmicroservices::::ServiceRegistration
    #include <cppmicroservices/ServiceRegistration.h> A registered service.

The framework returns a ServiceRegistration object when a BundleContext::RegisterService() method invocation is successful. The ServiceRegistration object is for the private use of the registering bundle and should not be shared with other bundles.

The ServiceRegistration object may be used to update the properties of the service or to unregister the service.

See BundleContext::RegisterService()

Template Parameters

• I1: Class type of the first service interface
• Interfaces: Template parameter pack containing zero or more service interfaces

Inherits from cppmicroservices::ServiceRegistrationBase

Public Functions

ServiceRegistration() Creates an invalid ServiceRegistration object.

You can use this object in boolean expressions and it will evaluate to false.

template <class Interface>
ServiceReference<Interface> GetReference() const

Returns a ServiceReference object for a service being registered.

The ServiceReference object may be shared with other bundles.

Return ServiceReference object.

Exceptions
• std::logic_error: If this ServiceRegistration object has already been unregistered or if it is invalid.

ServiceReference<I1> GetReference() const
Returns a ServiceReference object for a service being registered.

The ServiceReference object refers to the first interface type and may be shared with other bundles.

Return ServiceReference object.

Exceptions
• std::logic_error: If this ServiceRegistration object has already been unregistered or if it is invalid.

class cppmicroservices::ServiceRegistrationBase
#include <cppmicroservices/ServiceRegistrationBase.h> A registered service.

The framework returns a ServiceRegistrationBase object when a BundleContext::RegisterService() method invocation is successful. The ServiceRegistrationBase object is for the private use of the registering bundle and should not be shared with other bundles.

The ServiceRegistrationBase object may be used to update the properties of the service or to unregister the service.

Note: This class is provided as public API for low-level service management only. In almost all cases you should use the template ServiceRegistration instead.

See BundleContext::RegisterService()

Subclassed by cppmicroservices::ServiceRegistration<I1, Interfaces>, cppmicroservices::ServiceRegistration<cppmicroservices::HttpServletRequest>

Public Functions

ServiceRegistrationBase (const ServiceRegistrationBase &reg)
ServiceRegistrationBase (ServiceRegistrationBase &&reg)
operator bool() const
A boolean conversion operator converting this ServiceRegistrationBase object to true if it is valid and to false otherwise.

A ServiceRegistrationBase object is invalid if it was default-constructed or was invalidated by assigning 0 to it.

See operator=(std::nullptr_t)

Return true if this ServiceRegistrationBase object is valid, false otherwise.

ServiceRegistrationBase &operator= (std::nullptr_t)
Releases any resources held or locked by this ServiceRegistrationBase and renders it invalid.

Return This ServiceRegistrationBase object.
~ServiceRegistrationBase()

ServiceReferenceBase GetReference (const std::string &interfaceId = std::string()) const
Returns a ServiceReferenceBase object for a service being registered.
The ServiceReferenceBase object may be shared with other bundles.

Return ServiceReference object.

Exceptions
  • std::logic_error: If this ServiceRegistrationBase object has already been un-registered or if it is invalid.

void SetProperties (const ServiceProperties &properties)
Updates the properties associated with a service.
The Constants::OBJECTCLASS and Constants::SERVICE_ID keys cannot be modified by this method.
These values are set by the framework when the service is registered in the environment.
The following steps are taken to modify service properties:
  1. The service’s properties are replaced with the provided properties.
  2. A service event of type ServiceEvent::SERVICE_MODIFIED is fired.

Parameters
  • properties: The properties for this service. See ServiceProperties for a list of standard service property keys. Changes should not be made to this object after calling this method. To update the service’s properties this method should be called again.

Exceptions
  • std::logic_error: If this ServiceRegistrationBase object has already been un-registered or if it is invalid.
  • std::invalid_argument: If properties contains case variants of the same key name or if the number of the keys of properties exceeds the value returned by std::numeric_limits<int>::max().

void Unregister ()
Unregisters a service.
Remove a ServiceRegistrationBase object from the framework service registry. All ServiceRegistrationBase objects associated with this ServiceRegistrationBase object can no longer be used to interact with the service once unregistration is complete.
The following steps are taken to unregister a service:
  1. The service is removed from the framework service registry so that it can no longer be obtained.
  2. A service event of type ServiceEvent::SERVICE_UNREGISTERING is fired so that bundles using this service can release their use of the service. Once delivery of the service event is complete, the ServiceRegistrationBase objects for the service may no longer be used to get a service object for the service.
  3. For each bundle whose use count for this service is greater than zero: The bundle’s use count for this service is set to zero. If the service was registered with a ServiceFactory object, the ServiceFactory::UngetService method is called to release the service object for the bundle.

Exceptions
• `std::logic_error`: If this `ServiceRegistrationBase` object has already been unregistered or if it is invalid.

```cpp
bool operator< (const ServiceRegistrationBase &o) const
    Compare two `ServiceRegistrationBase` objects.

    If both `ServiceRegistrationBase` objects are valid, the comparison is done using the underlying `ServiceReference` object. Otherwise, this `ServiceRegistrationBase` object is less than the other object if and only if this object is invalid and the other object is valid.

    Return true if this `ServiceRegistrationBase` object is less than the other object.

Parameters

• o: The `ServiceRegistrationBase` object to compare with.

bool operator==(const ServiceRegistrationBase &registration) const
    `ServiceRegistrationBase &operator=(const ServiceRegistrationBase &registration)`
    `ServiceRegistrationBase &operator=(ServiceRegistrationBase &&registration)`

Friends

friend gr_serviceregistration::std::hash< ServiceRegistrationBase >

```cpp
template<>
struct std::hash<cppmicroservices::ServiceRegistrationBase>
    #include <cppmicroservices/ServiceRegistrationBase.h> Hash functor specialization for `ServiceRegistrationBase` objects.
```

15.1.22 ServiceTracker

```cpp
template <class S, class T = S>
class cppmicroservices::ServiceTracker
    #include <cppmicroservices/ServiceTracker.h> The `ServiceTracker` class simplifies using services from the framework’s service registry.

A `ServiceTracker` object is constructed with search criteria and a `ServiceTrackerCustomizer` object. A `ServiceTracker` can use a `ServiceTrackerCustomizer` to customize the service objects to be tracked. The `ServiceTracker` can then be opened to begin tracking all services in the framework’s service registry that match the specified search criteria. The `ServiceTracker` correctly handles all of the details of listening to `ServiceEvents` and getting and ungetting services.

The `GetServiceReferences` method can be called to get references to the services being tracked. The `GetService` and `GetServices` methods can be called to get the service objects for the tracked service.

Customization of the services to be tracked requires the tracked type to be default constructible and convertible to `bool`. To customize a tracked service using a custom type with value-semantics like

```cpp
struct MyTrackedClass
{
    explicit operator bool() const { return true; }
    /* ... */
};
```
a custom `ServiceTrackerCustomizer` is required. It provides code to associate the tracked service with the custom tracked type:

```cpp
struct MyTrackingCustomizer
    : public ServiceTrackerCustomizer<IFooService, MyTrackedClass>
{
    virtual std::shared_ptr<MyTrackedClass> AddingService(
        const ServiceReference<IFooService>&)
    {
        return std::shared_ptr<MyTrackedClass>();
    }

    virtual void ModifiedService(const ServiceReference<IFooService>&,
        const std::shared_ptr<MyTrackedClass>&)
    {}

    virtual void RemovedService(const ServiceReference<IFooService>&,
        const std::shared_ptr<MyTrackedClass>&)
    {}
};
```

Instantiation of a `ServiceTracker` with the custom customizer looks like this:

```cpp
MyTrackingCustomizer myCustomizer;
ServiceTracker<IFooService, MyTrackedClass> tracker(GetBundleContext(),
    &myCustomizer);
```

**Note**  The `ServiceTracker` class is thread-safe. It does not call a `ServiceTrackerCustomizer` while holding any locks. `ServiceTrackerCustomizer` implementations must also be thread-safe.

**Remark**  This class is thread safe.

**Template Parameters**

- S: The type of the service being tracked. The type S* must be an assignable datatype.
- T: The tracked object.

Inherits from `cppmicroservices::ServiceTrackerCustomizer<S, T>`

**Public Types**

```cpp
template<>
using TrackedParamType = typename ServiceTrackerCustomizer::TrackedParamType
    The type of the tracked object.
```

```cpp
template<>
using TrackingMap = std::unordered_map<ServiceReference<S>, std::shared_ptr<TrackedParamType>>
```

**Public Functions**

```cpp
~ServiceTracker()
    Automatically closes the `ServiceTracker`
```
ServiceTracker (const BundleContext &context, const ServiceReference<S> &reference, ServiceTrackerCustomizer<S, T> *customizer = nullptr)
Create a ServiceTracker on the specified ServiceReference.

The service referenced by the specified ServiceReference will be tracked by this ServiceTracker.

Parameters

- context: The BundleContext against which the tracking is done.
- reference: The ServiceReference for the service to be tracked.
- customizer: The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

ServiceTracker (const BundleContext &context, const std::string &clazz, ServiceTrackerCustomizer<S, T> *customizer = nullptr)
Create a ServiceTracker on the specified class name.

Services registered under the specified class name will be tracked by this ServiceTracker.

Parameters

- context: The BundleContext against which the tracking is done.
- clazz: The class name of the services to be tracked.
- customizer: The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

ServiceTracker (const BundleContext &context, const LDAPFilter &filter, ServiceTrackerCustomizer<S, T> *customizer = nullptr)
Create a ServiceTracker on the specified LDAPFilter object.

Services which match the specified LDAPFilter object will be tracked by this ServiceTracker.

Parameters

- context: The BundleContext against which the tracking is done.
- filter: The LDAPFilter to select the services to be tracked.
- customizer: The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

ServiceTracker (const BundleContext &context, ServiceTrackerCustomizer<S, T> *customizer = nullptr)
Create a ServiceTracker on the class template argument S.

Services registered under the interface name of the class template argument S will be tracked by this ServiceTracker.

Parameters
• **context**: The BundleContext against which the tracking is done.

• **customizer**: The customizer object to call when services are added, modified, or removed in this ServiceTracker. If customizer is null, then this ServiceTracker will be used as the ServiceTrackerCustomizer and this ServiceTracker will call the ServiceTrackerCustomizer methods on itself.

**Exceptions**

• **ServiceException**: If the service interface name is empty.

```cpp
global void Open()
```

Open this ServiceTracker and begin tracking services.

Services which match the search criteria specified when this ServiceTracker was created are now tracked by this ServiceTracker.

**Exceptions**

• std::runtime_error: If the BundleContext with which this ServiceTracker was created is no longer valid.

```cpp
global void Close()
```

Close this ServiceTracker.

This method should be called when this ServiceTracker should end the tracking of services.

This implementation calls GetServiceReferences() to get the list of tracked services to remove.

**Exceptions**

• std::runtime_error: If the BundleContext with which this ServiceTracker was created is no longer valid.

```cpp
std::shared_ptr<TrackedParamType> WaitForService()
```

Wait for at least one service to be tracked by this ServiceTracker.

This method will also return when this ServiceTracker is closed.

It is strongly recommended that WaitForService is not used during the calling of the BundleActivator methods. BundleActivator methods are expected to complete in a short period of time.

This implementation calls GetService() to determine if a service is being tracked.

**Return** The result of GetService().
This implementation calls \texttt{GetService()} to determine if a service is being tracked.

**Return** The result of \texttt{GetService()}.

**Parameters**
- \textit{rel\_time}: The relative time duration to wait for a service. If zero, the method will wait indefinitely.

**Exceptions**
- \texttt{std::invalid\_argument}: exception if \textit{rel\_time} is negative.

\texttt{virtual std::vector<ServiceReference<S>> GetServiceReferences () const}

Return a list of \texttt{ServiceReference}s for all services being tracked by this \texttt{ServiceTracker}.

**Return** A list of \texttt{ServiceReference} objects.

\texttt{virtual ServiceReference<S> GetServiceReference () const}

Returns a \texttt{ServiceReference} for one of the services being tracked by this \texttt{ServiceTracker}.

If multiple services are being tracked, the service with the highest ranking (as specified in its \texttt{service\_ranking} property) is returned. If there is a tie in ranking, the service with the lowest service ID (as specified in its \texttt{service\_id} property); that is, the service that was registered first is returned. This is the same algorithm used by \texttt{BundleContext::GetServiceReference()}.

This implementation calls \texttt{GetServiceReferences()} to get the list of references for the tracked services.

**Return** A \texttt{ServiceReference} for a tracked service.

**Exceptions**
- \texttt{ServiceException}: if no services are being tracked.

\texttt{virtual std::shared\_ptr<TrackedParamType> GetService (const ServiceReference<S> &reference) const}

Returns the service object for the specified \texttt{ServiceReference} if the specified referenced service is being tracked by this \texttt{ServiceTracker}.

**Return** A service object or \texttt{nullptr} if the service referenced by the specified \texttt{ServiceReference} is not being tracked.

**Parameters**
- \textit{reference}: The reference to the desired service.

\texttt{virtual std::vector<std::shared\_ptr<TrackedParamType>> GetServices () const}

Return a list of service objects for all services being tracked by this \texttt{ServiceTracker}.

This implementation calls \texttt{GetServiceReferences()} to get the list of references for the tracked services and then calls \texttt{GetService(const ServiceReference &)} for each reference to get the tracked service object.

**Return** A list of service objects or an empty list if no services are being tracked.

\texttt{virtual std::shared\_ptr<TrackedParamType> GetService () const}

Returns a service object for one of the services being tracked by this \texttt{ServiceTracker}.

If any services are being tracked, this implementation returns the result of calling \texttt{GetService(GetServiceReference())}. 

---

Chapter 15. Framework API
Return a service object or null if no services are being tracked.

virtual void Remove (const ServiceReference<S> &reference)
Remove a service from this ServiceTracker.

The specified service will be removed from this ServiceTracker. If the specified service was being tracked then the ServiceTrackerCustomizer::RemovedService method will be called for that service.

Parameters
• reference: The reference to the service to be removed.

virtual int Size () const
Return the number of services being tracked by this ServiceTracker.

Return The number of services being tracked.

virtual int GetTrackingCount () const
Returns the tracking count for this ServiceTracker.

The tracking count is initialized to 0 when this ServiceTracker is opened. Every time a service is added, modified or removed from this ServiceTracker, the tracking count is incremented.

The tracking count can be used to determine if this ServiceTracker has added, modified or removed a service by comparing a tracking count value previously collected with the current tracking count value. If the value has not changed, then no service has been added, modified or removed from this ServiceTracker since the previous tracking count was collected.

Return The tracking count for this ServiceTracker or -1 if this ServiceTracker is not open.

virtual void GetTracked (TrackingMap &tracked) const
Return a sorted map of the ServiceReferences and service objects for all services being tracked by this ServiceTracker.

The map is sorted in natural order of ServiceReference. That is, the last entry is the service with the highest ranking and the lowest service id.

Parameters
• tracked: A TrackingMap with the ServiceReferences and service objects for all services being tracked by this ServiceTracker. If no services are being tracked, then the returned map is empty.

virtual bool IsEmpty () const
Return if this ServiceTracker is empty.

Return true if this ServiceTracker is not tracking any services.

Protected Functions

std::shared_ptr<TrackedParamType> AddingService (const ServiceReference<S> &reference)
Default implementation of the ServiceTrackerCustomizer::AddingService method.
This method is only called when this `ServiceTracker` has been constructed with a `nullptr` `ServiceTrackerCustomizer` argument.

This implementation returns the result of calling `GetService` on the `BundleContext` with which this `ServiceTracker` was created passing the specified `ServiceReference`.

This method can be overridden in a subclass to customize the service object to be tracked for the service being added. In that case, take care not to rely on the default implementation of `RemovedService` to unget the service.

**Return**  The service object to be tracked for the service added to this `ServiceTracker`.

**See**  `ServiceTrackerCustomizer::AddingService(const ServiceReference&)`

**Parameters**

- `reference`: The reference to the service being added to this `ServiceTracker`.

**Exceptions**

- `std::runtime_error`: If this `BundleContext` is no longer valid.
- `std::invalid_argument`: If the specified `ServiceReference` is invalid (default constructed).

```cpp
void ModifiedService(const ServiceReference<S> &reference, const std::shared_ptr<TrackedParamType> &service)
```

Default implementation of the `ServiceTrackerCustomizer::ModifiedService` method.

This method is only called when this `ServiceTracker` has been constructed with a `nullptr` `ServiceTrackerCustomizer` argument.

This implementation does nothing.

**See**  `ServiceTrackerCustomizer::ModifiedService(const ServiceReference&, TrackedArgType)`

**Parameters**

- `reference`: The reference to modified service.
- `service`: The service object for the modified service.

```cpp
void RemovedService(const ServiceReference<S> &reference, const std::shared_ptr<TrackedParamType> &service)
```

Default implementation of the `ServiceTrackerCustomizer::RemovedService` method.

This method is only called when this `ServiceTracker` has been constructed with a `nullptr` `ServiceTrackerCustomizer` argument.

This method can be overridden in a subclass. If the default implementation of the `AddingService` method was used, this method must unget the service.

**See**  `ServiceTrackerCustomizer::RemovedService(const ServiceReferenceType&, TrackedArgType)`

**Parameters**

- `reference`: The reference to removed service.
- `service`: The service object for the removed service.

```cpp
template <class S, class T = S>
```
struct cppmicroservices::ServiceTrackerCustomizer

#include <cppmicroservices/ServiceTrackerCustomizer.h> The ServiceTrackerCustomizer interface allows a ServiceTracker to customize the service objects that are tracked.

A ServiceTrackerCustomizer is called when a service is being added to a ServiceTracker. The ServiceTrackerCustomizer can then return an object for the tracked service. A ServiceTrackerCustomizer is also called when a tracked service is modified or has been removed from a ServiceTracker.

The methods in this interface may be called as the result of a ServiceEvent being received by a ServiceTracker. Since ServiceEvents are synchronously delivered, it is highly recommended that implementations of these methods do not register (BundleContext::RegisterService), modify (ServiceRegistration::SetProperties) or unregister (ServiceRegistration::Unregister) a service while being synchronized on any object.

Note The ServiceTracker class implementation of ServiceTrackerCustomizer is thread-safe. It does not call a ServiceTrackerCustomizer while holding any locks. ServiceTrackerCustomizer implementations must also be thread-safe.

Remark ServiceTrackerCustomizer implementations must also be thread-safe.

Template Parameters
- \( S \): The type of the service being tracked
- \( T \): The type of the tracked object. The default is \( S \).

Subclassed by cppmicroservices::ServiceTracker< \( S, T \) >

Public Types

\[ \text{template<> using TrackedParamType = typename TypeTraits::TrackedParamType} \]

Public Functions

virtual ~ServiceTrackerCustomizer()

virtual std::shared_ptr<TrackedParamType> AddingService(const ServiceReference<\( S \)>& reference) = 0

A service is being added to the ServiceTracker.

This method is called before a service which matched the search parameters of the ServiceTracker is added to the ServiceTracker. This method should return the service object to be tracked for the specified ServiceReference. The returned service object is stored in the ServiceTracker and is available from the getService and getServices methods.

Return The service object to be tracked for the specified referenced service or 0 if the specified referenced service should not be tracked.

Parameters
- reference: The reference to the service being added to the ServiceTracker.
virtual void ModifiedService (const ServiceReference<S> &reference, const std::shared_ptr<TrackedParamType> &service) = 0

A service tracked by the ServiceTracker has been modified.

This method is called when a service being tracked by the ServiceTracker has had its properties modified.

Parameters

• reference: The reference to the service that has been modified.
• service: The service object for the specified referenced service.

virtual void RemovedService (const ServiceReference<S> &reference, const std::shared_ptr<TrackedParamType> &service) = 0

A service tracked by the ServiceTracker has been removed.

This method is called after a service is no longer being tracked by the ServiceTracker.

Parameters

• reference: The reference to the service that has been removed.
• service: The service object for the specified referenced service.

struct TypeTraits

#include <cppmicroservices/ServiceTrackerCustomizer.h>

Public Types

template<>
using ServiceType = S
template<>
using TrackedType = T
template<>
using TrackedParamType = T

Public Static Functions

template<>
static std::shared_ptr<TrackedType> ConvertToTrackedType (const std::shared_ptr<S>&)

15.2 Utilities

These classes support the main CppMicroServices API:

15.2.1 Any

template <typename ValueType>
Value *cppmicroservices::any_cast (Any *operand)
any_cast operator used to extract the ValueType from an Any*.
Will return a pointer to the stored value.
Example Usage:

```cpp
defaultt pTmp = any_cast<defaultt>(pAny)
```

Will return nullptr if the cast fails, i.e. types don’t match.

```
template <typename ValueType>
const ValueType *cppmicroservices::any_cast (const Any *operand)
any_cast operator used to extract a const ValueType pointer from a const Any*.
Will return a const pointer to the stored value.
Example Usage:

```cpp
defaultt pTmp = any_cast<defaultt>(pAny)
```

Will return nullptr if the cast fails, i.e. types don’t match.

```
template <typename ValueType>
ValueType cppmicroservices::any_cast (const Any &operand)
any_cast operator used to extract a copy of the ValueType from a const Any&.
Example Usage:

```cpp
default tmp = any_cast<default>(anAny)
```

Don’t use an any_cast in combination with references, i.e. MyType& tmp = ... or const MyType& = ... Some compilers will accept this code although a copy is returned. Use the ref_any_cast in these cases.

Exceptions

- **BadAnyCastException**: if the cast fails.

```
template <typename ValueType>
ValueType cppmicroservices::any_cast (Any &operand)
any_cast operator used to extract a copy of the ValueType from an Any&.
Example Usage:

```cpp
default tmp = any_cast<default>(anAny)
```

Don’t use an any_cast in combination with references, i.e. MyType& tmp = ... or const MyType& tmp = ... Some compilers will accept this code although a copy is returned. Use the ref_any_cast in these cases.

Exceptions

- **BadAnyCastException**: if the cast fails.

```
template <typename ValueType>
const ValueType &cppmicroservices::ref_any_cast (const Any &operand)
ref_any_cast operator used to return a const reference to the internal data.
Example Usage:

```cpp
const MyType& tmp = ref_any_cast<MyType>(anAny);
```

Exceptions

- **BadAnyCastException**: if the cast fails.
template <typename ValueType>
ValueType &cppmicroservices::ref_any_cast(Any &operand)
ref_any_cast operator used to return a reference to the internal data.

Example Usage:

MyType& tmp = ref_any_cast<MyType>(anAny);

Exceptions

• BadAnyCastException: if the cast fails.

class cppmicroservices::Any

#include <cppmicroservices/Any.h> An Any class represents a general type and is capable of storing any type, supporting type-safe extraction of the internally stored data.


Public Functions

Any ()
Creates an empty any type.

template <typename ValueType>
Any (const ValueType &value)
Creates an Any which stores the init parameter inside.

Example:

Any a(13);
Any a(string("12345"));

Parameters

• value: The content of the Any

Any (const Any &other)
Copy constructor, works with empty Anys and initialized Any values.

Parameters

• other: The Any to copy

Any (Any &&other)
Move constructor.

Parameters

• other: The Any to move

Any &Swap (Any &rhs)
Swaps the content of the two Anys.

Parameters
template <typename ValueType>
bool operator==(const ValueType &val) const

Compares this Any with another value.

If the internal type of this any and of val do not match, the comparison always returns false.

Return true if this Any contains value val, false otherwise.

Parameters

- val: The value to compare to.

bool operator==(const Any &rhs) const

Compares this Any with another Any.

We accomplish this by forwarding the call to a virtual compare function on the Holder of the value in the _content field. The Placeholder subclass of Holder provides an implementation that invokes the above operator== with the underlying value of ValueType.

Return bool return true if rhs compares equal to *this AND the underlying ValueType has an operator==, and return false otherwise.

Parameters

- rhs: an Any to compare against

template <typename ValueType>
bool operator!=(const ValueType &val) const

Compares this Any with another value for inequality.

This is the same as

!this->operator==(val)

Return true if this Any does not contain value val, false otherwise.

Parameters

- val: The value to compare to.

template <typename ValueType>
Any &operator=(const ValueType &rhs)

Assignment operator for all types != Any.

Example:

Any a = 13;
Any a = string("12345");

Parameters

- rhs: The value which should be assigned to this Any.

Any &operator=(const Any &rhs)

Assignment operator for Any.

Parameters

- rhs: The Any which should be assigned to this Any.
Move assignment operator for `Any`.

**Return** A reference to this `Any`.

**Parameters**

- `rhs`: The `Any` which should be moved into this `Any`.

**Exceptions**

- `std::logic_error`: if the `Any` is empty.

Returns a string representation for the content if it is not empty.

Custom types should either provide a `std::ostream& operator<<(std::ostream& os, const CustomType& ct)` function or specialize the `any_value_to_string` template function for meaningful output.

**Exceptions**

- `std::logic_error`: if the `Any` is empty.

Returns a string representation for the content.

If the `Any` is empty, an empty string is returned.

Custom types should either provide a `std::ostream& operator<<(std::ostream& os, const CustomType& ct)` function or specialize the `any_value_to_string` template function for meaningful output.

Returns a JSON representation for the content.

Custom types should specialize the `any_value_to_json` template function for meaningful output. The values of `increment` and `indent` are passed around to be able to be used for nicer formatting. The code that makes use of this is in the `any_value_to_json` specializations for the various containers.

To get pretty output, simply pass a value greater than zero in as the first argument of `ToJSON` and the rest of the code will take care of things.

**Parameters**

- `increment`: The amount of extra indentation to add for each level of JSON. An increment of zero indicates no special formatting
- `indent`: The current amount of indent to apply to the current line.

Returns the type information of the stored content.

If the `Any` is empty `typeid(void)` is returned. It is suggested to always query an `Any` for its type info before trying to extract data via an `any_cast/ref_any_cast`.
class cppmicroservices::BadAnyCastException
#include <cppmicroservices/Any.h> The BadAnyCastException class is thrown in case of casting an Any instance.
Inherits from std::bad_cast

Public Functions

BadAnyCastException (std::string msg = "")
~BadAnyCastException ()
const char *what () const

15.2.2 AnyMap

class cppmicroservices::AnyMap
A map data structure with support for compound keys.
This class adds convenience functions on top of the any_map class. The any_map is a recursive data structure, and its values can be retrieved via standard map functions or by using a dotted key notation specifying a compound key.

See any_map
Inherits from cppmicroservices::any_map

Public Functions

AnyMap (map_type type)
AnyMap (const ordered_any_map &m)
AnyMap (ordered_any_map &&m)
AnyMap (const unordered_any_map &m)
AnyMap (unordered_any_map &&m)
AnyMap (const unordered_any_cimap &m)
AnyMap (unordered_any_cimap &&m)
map_type GetType () const
Get the underlying STL container type.

Return The STL container type holding the map data.

cst mapped_type &AtCompoundKey (const key_type &key) const
Get a key’s value, using a compound key notation.
A compound key consists of one or more key names, concatenated with the ‘.’ (dot) character. Each key except the last requires the referenced Any object to be of type AnyMap or std::vector<Any>. Containers of type std::vector<Any> are indexed using 0-based numerical key names.
For example, a `AnyMap` object holding data of the following layout

```cpp
{
    one: 1,
    two: "two",
    three: {
        a: "anton",
        b: [3, 8]
    }
}
```

can be queried using the following notation:

```cpp
class AnyMap {
    // ... methods...

    Any AtCompoundKey(const std::string& key) const {
        // ... implementation...
        return value;  // return Any(1)
    }
}
```

Return: A reference to the key’s value.

Parameters:
- `key`: The key hierarchy to query.

Exceptions:
- `std::invalid_argument`: if the `Any` value for a given key is not of type `AnyMap` or `std::vector<Any>`.
- `std::out_of_range`: if the key is not found or a numerical index would fall out of the range of an int type.

```
map.AtCompoundKey("one");  // returns Any(1)
map.AtCompoundKey("three.a");  // returns Any(std::string("anton"))
map.AtCompoundKey("three.b.1");  // returns Any(8)
```

A compound key consists of one or more key names, concatenated with the ‘.’ (dot) character. Each key except the last requires the referenced `Any` object to be of type `AnyMap` or `std::vector<Any>`. Containers of type `std::vector<Any>` are indexed using 0-based numerical key names.

For example, a `AnyMap` object holding data of the following layout

```cpp
{
    one: 1,
    two: "two",
    three: {
        a: "anton",
        b: [3, 8]
    }
}
```

can be queried using the following notation:

```
map.AtCompoundKey("one", Any());  // returns Any(1)
map.AtCompoundKey("four", Any());  // returns Any()
map.AtCompoundKey("three.a", Any());  // returns Any(std::string("anton"))
map.AtCompoundKey("three.c", Any());  // returns Any()
map.AtCompoundKey("three.b.1", Any());  // returns Any(8)
map.AtCompoundKey("three.b.4", Any());  // returns Any()
```
Return A copy of the key’s value.

Parameters

- key: The key hierarchy to query.
- defaultValue: is the value to be returned if the key is not found

class cppmicroservices::any_map
A map data structure which wraps different STL map types.

This is a convenience class providing a STL associative container interface for different underlying container types. Supported underlying types are

- `any_map::ordered_any_map` (a STL map)
- `any_map::unordered_any_map` (a STL unordered map)
- `any_map::unordered_any_cimap` (a STL unordered map with case insensitive key comparison)

This class provides most of the STL functions for associated containers, including forward iterators. It is typically not instantiated by clients directly, but obtained via framework API calls, returning an `AnyMap` sub-class instance.

See `AnyMap`

Subclassed by `cppmicroservices::AnyMap`

Public Types

class enum map_type
Values:

- ORDERED_MAP
- UNORDERED_MAP
- UNORDERED_MAP_CASEINSENSITIVE_KEYS

typedef using key_type = std::string
typedef using mapped_type = Any
typedef using value_type = std::pair<const key_type, mapped_type>
typedef using size_type = std::size_t
typedef using difference_type = std::ptrdiff_t
typedef using reference = value_type&
typedef using const_reference = const value_type&
typedef using pointer = value_type*
typedef using const_pointer = const value_type*
typedef using ordered_any_map = std::map<std::string, Any>
typedef using unordered_any_map = std::unordered_map<std::string, Any>
typedef using unordered_any_cimap = std::unordered_map<std::string, Any, detail::any_map_cihash, detail::any_map_ciequal>
typedef using iterator = iter
using const_iterator = const_iter

Public Functions

any_map(map_type type)
any_map(const ordered_any_map &m)
any_map(ordered_any_map &m)
any_map(const unordered_any_map &m)
any_map(unordered_any_map &m)
any_map(const unordered_any_cimap &m)
any_map(unordered_any_cimap &m)
any_map(const any_map &m)

any_map &operator=(const any_map &m)

any_map(any_map &&m)

any_map &operator=(any_map &&m)

~any_map()

iter begin()
const_iter begin() const
const_iter cbegin() const
iter end()
const_iter end() const
const_iter cend() const
bool empty() const
size_type size() const
size_type count(const key_type &key) const
void clear()

mapped_type &at(const key_type &key)

const mapped_type &at(const key_type &key) const
mapped_type &operator[] (const key_type &key)
mapped_type &operator[] (key_type &&key)

std::pair<iterator, bool> insert(const value_type &value)

template <class... Args>
std::pair<iterator, bool> emplace(Args&&... args)
const_iterator find(const key_type &key) const
return the iterator to the value referenced by key

size_type erase(const key_type &key)
Erase entry for value for ‘key’.

Return the number of elements erased.

Parameters

• key: the key for the entry to Erase

bool operator==(const any_map &rhs) const
Compare the content of this map with those of rhs.

Return bool true rhs contains the same content as this

Parameters

• rhs: an any_map to compare

bool operator!=(const any_map &rhs) const

Public Members

ordered_any_map *o
unordered_any_map *uo
unordered_any_cimap *uoci

Protected Attributes

map_type type

class const_iter
Inherits from cppmicroservices::any_map::iterator_base

Public Types

using reference = any_map::const_reference
using pointer = any_map::const_pointer
using iterator = const_iter

Public Functions

const_iter()
const_iter(const iterator &it)
const_iter(const iter &it)
~const_iter()
const_iter (ociter &it)
const_iter (uociter &it, iter_type type)

reference operator*() const
pointer operator->() const

iterator &operator++()
iterator operator++(int)

bool operator==(const iterator &x) const
bool operator!=(const iterator &x) const

Public Members

ociter *o
uociter *uo
uocciiter *uoci

class iter
Inherits from cppmicroservices::any_map::iterator_base

Public Types

using reference = any_map::reference
using pointer = any_map::pointer
using iterator = iter

Public Functions

iter()
iter(const iter &it)
~iter()
iter(oiter &it)
iter(uoiter &it, iter_type type)

reference operator*() const
pointer operator->() const

iterator &operator++()
iterator operator++(int)

bool operator==(const iterator &x) const
bool operator!=(const iterator &x) const
Public Members

oiter *o
uoiter *uo
uociiter *uoci

15.2.3 BundleVersion

std::ostream &cppmicroservices::operator<<(std::ostream &os, const BundleVersion &v)
Streams the string representation of v into the stream os, using BundleVersion::ToString.

class cppmicroservices::BundleVersion
#include <cppmicroservices/BundleVersion.h> Version identifier for CppMicroServices bundles.

Version identifiers have four components.
4. Qualifier. A text string. See BundleVersion(const std::string&) for the format of the qualifier string.

BundleVersion objects are immutable.

Public Functions

BundleVersion (unsigned int majorVersion, unsigned int minorVersion, unsigned int microVersion)
Creates a version identifier from the specified numerical components.

The qualifier is set to the empty string.

Parameters

• majorVersion: Major component of the version identifier.
• minorVersion: Minor component of the version identifier.
• microVersion: Micro component of the version identifier.

BundleVersion (unsigned int majorVersion, unsigned int minorVersion, unsigned int microVersion, std::string qualifier)
Creates a version identifier from the specified components.

Parameters

• majorVersion: Major component of the version identifier.
• minorVersion: Minor component of the version identifier.
• microVersion: Micro component of the version identifier.
• qualifier: Qualifier component of the version identifier.
BundleVersion (const std::string &version)
   Created a version identifier from the specified string.

   Here is the grammar for version strings.
   There must be no whitespace in version.

   Parameters
   • version: string representation of the version identifier.

BundleVersion (const BundleVersion &version)
   Create a version identifier from another.

   Parameters
   • version: Another version identifier

bool IsUndefined () const
   Returns the undefined state of this version identifier.

   Return true if this version identifier is undefined, false otherwise.

unsigned int GetMajor () const
   Returns the majorVersion component of this version identifier.

   Return The majorVersion component.

unsigned int GetMinor () const
   Returns the minorVersion component of this version identifier.

   Return The minorVersion component.

unsigned int GetMicro () const
   Returns the microVersion component of this version identifier.

   Return The microVersion component.

std::string GetQualifier () const
   Returns the qualifier component of this version identifier.

   Return The qualifier component.

std::string ToString () const
   Returns the string representation of this version identifier.

   The format of the version string will be majorVersion.minorVersion.microVersion if qualifier is the empty string or majorVersion.minorVersion.microVersion.qualifier otherwise.

   Return The string representation of this version identifier.
bool operator==(const BundleVersion &object) const
    
    Compares this BundleVersion object to another object.

    A version is considered to be equal to another version if the majorVersion, minorVersion and microVersion
    components are equal and the qualifier component is equal.

    Return true if object is a BundleVersion and is equal to this object; false otherwise.

    Parameters
      • object: The BundleVersion object to be compared.

int Compare(const BundleVersion &object) const
    
    Compares this BundleVersion object to another object.

    A version is considered to be less than another version if its majorVersion component is less than the other
    version’s majorVersion component, or the majorVersion components are equal and its minorVersion com-
    ponent is less than the other version’s minorVersion component, or the majorVersion and minorVersion
    components are equal and its microVersion component is less than the other version’s microVersion com-
    ponent, or the majorVersion, minorVersion and microVersion components are equal and it’s qualifier com-
    ponent is less than the other version’s qualifier component (using std::string::operator<()).

    A version is considered to be equal to another version if the majorVersion, minorVersion and microVersion
    components are equal and the qualifier component is equal.

    Return A negative integer, zero, or a positive integer if this object is less than, equal to, or greater than
    the specified BundleVersion object.

    Parameters
      • object: The BundleVersion object to be compared.

Public Static Functions

static BundleVersion EmptyVersion()
    
    The empty version “0.0.0”.

static BundleVersion UndefinedVersion()
    
    Creates an undefined version identifier, representing either infinity or minus infinity.

static BundleVersion ParseVersion(const std::string &version)
    
    Parses a version identifier from the specified string.

    See BundleVersion(const std::string&) for the format of the version string.

    Return A BundleVersion object representing the version identifier. If version is the empty string
    then EmptyVersion will be returned.

    Parameters
      • version: string representation of the version identifier. Leading and trailing whitespace will be
        ignored.
15.2.4 Constants

namespace cppmicroservices::Constants

Defines standard names for the CppMicroServices environment system properties, service properties, and Manifest header attribute keys.

The values associated with these keys are of type std::string, unless otherwise indicated.

Variables

const std::string SYSTEM_BUNDLE_LOCATION

Location identifier of the OSGi system bundle, which is defined to be “System Bundle”.

const std::string SYSTEM_BUNDLE_SYMBOLICNAME

Alias for the symbolic name of the OSGi system bundle.

It is defined to be “system.bundle”.

const std::string BUNDLE_ACTIVATOR

Manifest header identifying the bundle’s activator.

The value for this attribute is of type bool. false - the bundle has no activator true - the bundle has an activator The behavior if the attribute is not specified is the same as when it is set to ‘false’.

The header value may be retrieved via the Bundle::GetProperty method.

const std::string BUNDLE_CATEGORY

Manifest header identifying the bundle’s category.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_COPYRIGHT

Manifest header identifying the bundle’s copyright information.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_DESCRIPTION

Manifest header containing a brief description of the bundle’s functionality.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_MANIFESTVERSION

Manifest header identifying the bundle’s manifest version.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_NAME

Manifest header identifying the bundle’s name.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_VENDOR

Manifest header identifying the bundle’s vendor.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.
const std::string BUNDLE_VERSION
Manifest header identifying the bundle’s version.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_DOCURL
Manifest header identifying the bundle’s documentation URL, from which further information about the bundle may be obtained.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_CONTACTADDRESS
Manifest header identifying the contact address where problems with the bundle may be reported; for example, an email address.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_SYMBOLICNAME
Manifest header identifying the bundle’s symbolic name.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

const std::string BUNDLE_LOCALIZATION
Manifest header identifying the base name of the bundle’s localization entries.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

See BUNDLE_LOCALIZATION_DEFAULT_BASENAME

const std::string BUNDLE_LOCALIZATION_DEFAULT_BASENAME
Default value for the bundle.localization manifest header.

See BUNDLE_LOCALIZATION

const std::string BUNDLE_ACTIVATIONPOLICY
Manifest header identifying the bundle’s activation policy.

The header value may be retrieved from the AnyMap object returned by the Bundle::GetHeaders() method.

See ACTIVATION_LAZY

const std::string ACTIVATION_LAZY
Bundle activation policy declaring the bundle must be activated when the library containing it is loaded into memory.

A bundle with the lazy activation policy that is started with the START_ACTIVATION_POLICY option will wait in the STATE_STARTING state until its library is loaded. The bundle will then be activated.

The activation policy value is specified as in the bundle.activation_policy manifest header like:

See BUNDLE_ACTIVATIONPOLICY
See Bundle::Start(uint32_t)
See Bundle::START_ACTIVATION_POLICY

15.2. Utilities 175
const std::string FRAMEWORK_VERSION
Framework environment property identifying the Framework version.
The header value may be retrieved via the BundleContext::GetProperty method.

const std::string FRAMEWORK_VENDOR
Framework environment property identifying the Framework implementation vendor.
The header value may be retrieved via the BundleContext::GetProperty method.

const std::string FRAMEWORK_STORAGE
Framework launching property specifying the persistent storage area used by the framework.
The value of this property must be a valid file path in the file system to a directory. If the specified directory
does not exist then the framework will create the directory. If the specified path exists but is not a directory
or if the framework fails to create the storage directory, then framework initialization fails. This area can
not be shared with anything else.

If this property is not set, the framework uses the “fwdir” directory in the current working directory for the
persistent storage area.

const std::string FRAMEWORK_STORAGE_CLEAN
Framework launching property specifying if and when the persistent storage area for the framework should
be cleaned.

If this property is not set, then the framework storage area must not be cleaned.

See FRAMEWORK_STORAGE_CLEAN_ONFIRSTINIT

const std::string FRAMEWORK_STORAGE_CLEAN_ONFIRSTINIT
Specifies that the framework storage area must be cleaned before the framework is initialized for the first
time.

Subsequent inits, starts or updates of the framework will not result in cleaning the framework storage area.

const std::string FRAMEWORK_THREADING_SUPPORT
The framework’s threading support property key name.
This property’s default value is “single”. Valid key values are:
• “single” - The framework APIs are not thread-safe.
• “multi” - The framework APIs are thread-safe.

Note: This is a read-only property and cannot be altered at run-time. The key’s value is set at compile
time by the US_ENABLE_THREADING_SUPPORT option. See Build Instructions for more information.

const std::string FRAMEWORK_THREADING_SINGLE
Framework threading support configuration declaring that the framework is configured for single thread
usage.

It is not safe to use the framework API concurrently from multiple threads.

const std::string FRAMEWORK_THREADING_MULTI
Framework threading support configuration declaring that the framework is configured for multi-thread
usage.

The framework API uses internal locks such that it is safe to use the API concurrently from multiple threads.
const std::string FRAMEWORK_LOG
    The framework’s log property key name.

    This property’s default value is off (boolean ‘false’).

cost std::string FRAMEWORK_UUID
    Framework environment property identifying the Framework’s universally unique identifier (UUID).

    A UUID represents a 128-bit value. A new UUID is generated by the Framework::Init() method each time
    a framework is initialized. The value of this property conforms to the UUID string representation specified
    in RFC 4122.

    The header value may be retrieved via the BundleContext::GetProperty method.

cost std::string FRAMEWORK_WORKING_DIR
    Framework launching property specifying the working directory used for resolving relative path names.

    If not set, the framework will use the process current working directory as set during static initialization of
    the framework library.

cost std::string FRAMEWORK_BUNDLE_VALIDATION_FUNC
    Framework bundle validation property specifying a function that control whether a shared library is loaded
    or not into the process.

cost std::string OBJECTCLASS
    Service property identifying all of the class names under which a service was registered in the Framework.

    The value of this property must be of type std::vector<std::string>.

    This property is set by the Framework when a service is registered.

cost std::string SERVICE_ID
    Service property identifying a service’s registration number.

    The value of this property must be of type long int.

    The value of this property is assigned by the Framework when a service is registered. The Framework
    assigns a unique value that is larger than all previously assigned values since the Framework was started.
    These values are NOT persistent across restarts of the Framework.

cost std::string SERVICE_PID
    Service property identifying a service’s persistent identifier.

    This property may be supplied in the properties ServiceProperties object passed to the
    BundleContext::RegisterService method. The value of this property must be of type
    std::string or std::vector<std::string>.

    A service’s persistent identifier uniquely identifies the service and persists across multiple Framework
    invocations.

    By convention, every bundle has its own unique namespace, starting with the bundle’s identifier (see
    Bundle::GetBundleId()) and followed by a dot (.). A bundle may use this as the prefix of the persistent
    identifiers for the services it registers.

cost std::string SERVICE_RANKING
    Service property identifying a service’s ranking number.

    This property may be supplied in the ServiceProperties object passed to the
    BundleContext::RegisterService method. The value of this property must be of type
    int.

    The service ranking is used by the framework to determine the natural order of services, see ServiceRef-
    erence::operator<(const ServiceReference&), and the default service to be returned from a call to the
    BundleContext::GetServiceReference method.
The default ranking is zero (0). A service with a ranking of \texttt{std::numeric_limits<int>::max()} is very likely to be returned as the default service, whereas a service with a ranking of \texttt{std::numeric_limits<int>::min()} is very unlikely to be returned.

If the supplied property value is not of type \texttt{int}, it is deemed to have a ranking value of zero.

c\texttt{onst std::string SERVICE_VENDOR}
Service property identifying a service’s vendor.

This property may be supplied in the properties \texttt{ServiceProperties} object passed to the \texttt{BundleContext::RegisterService} method.

c\texttt{onst std::string SERVICE_DESCRIPTION}
Service property identifying a service’s description.

This property may be supplied in the properties \texttt{ServiceProperties} object passed to the \texttt{BundleContext::RegisterService} method.

c\texttt{onst std::string SERVICE_SCOPE}
Service property identifying a service’s scope.

This property is set by the framework when a service is registered. If the registered object implements \texttt{PrototypeServiceFactory}, then the value of this service property will be \texttt{SCOPE_PROTOTYPE}. Otherwise, if the registered object implements \texttt{ServiceFactory}, then the value of this service property will be \texttt{SCOPE_BUNDLE}. Otherwise, the value of this service property will be \texttt{SCOPE_SINGLETON}.

c\texttt{onst std::string SCOPE_SINGLETON}
Service scope is singleton.

All bundles using the service receive the same service object.

\textbf{See} \texttt{SERVICE_SCOPE}

c\texttt{onst std::string SCOPE_BUNDLE}
Service scope is bundle.

Each bundle using the service receives a distinct service object.

\textbf{See} \texttt{SERVICE_SCOPE}

c\texttt{onst std::string SCOPE_PROTOTYPE}
Service scope is prototype.

Each bundle using the service receives either a distinct service object or can request multiple distinct service objects via \texttt{ServiceObjects}.

\textbf{See} \texttt{SERVICE_SCOPE}

c\texttt{onst std::string LIBRARY_LOAD_OPTIONS}
Service property that holds optional flags for \texttt{dlopen} calls on POSIX systems.

\subsection{15.2.5 FrameworkFactory}

c\texttt{lass cppmicroservices::FrameworkFactory}
A factory for creating \texttt{Framework} instances.

\textbf{Remark} This class is thread-safe.
Public Functions

Framework NewFramework(const FrameworkConfiguration &configuration, std::ostream *logger = nullptr)
Create a new Framework instance.

Return A new, configured Framework instance.

Parameters

- configuration: The framework properties to configure the new framework instance. If framework properties are not provided by the configuration argument, the created framework instance will use a reasonable default configuration.

- logger: Any ostream object which will receive redirected debug log output.

Framework NewFramework()
Create a new Framework instance.
This is the same as calling

NewFramework{FrameworkConfiguration{}}

Return A new, configured Framework instance.

Framework NewFramework(const std::map<std::string, Any> &configuration, std::ostream *logger = nullptr)
Create a new Framework instance.

Return A new, configured Framework instance.

15.2.6 GetBundleContext

static BundleContext cppmicroservices::GetBundleContext()
Returns the bundle context of the calling bundle.
This function allows easy access to the BundleContext instance from inside a C++ Micro Services bundle.

Return The BundleContext of the calling bundle. If the caller is not part of an active bundle, an invalid BundleContext is returned.

15.2.7 LDAPFilter

std::ostream &cppmicroservices::operator<<(std::ostream &os, const LDAPFilter &filter)
Streams the string representation of filter into the stream os via LDAPFilter::ToString().

LDAPPropExpr cppmicroservices::LDAPProp::operator==(const std::string &s) const

LDAPPropExpr cppmicroservices::LDAPProp::operator==(const cppmicroservices::Any &s) const

LDAPPropExpr cppmicroservices::LDAPProp::operator==(bool b) const

template <class T>
LDAPPropExpr cppmicroservices::LDAPProp::operator==(const T &s) const

15.2. Utilities 179
LDAPPropExpr cppmicroservices::LDAPProp::operator!= (const std::string & s) const

template <class T>
LDAPPropExpr cppmicroservices::LDAPProp::operator!= (const T & s) const

LDAPPropExpr cppmicroservices::LDAPProp::operator>=(const std::string & s) const

template <class T>
LDAPPropExpr cppmicroservices::LDAPProp::operator>=(const T & s) const

LDAPPropExpr cppmicroservices::LDAPProp::operator<=(const std::string & s) const

template <class T>
LDAPPropExpr cppmicroservices::LDAPProp::operator<=(const T & s) const

 LDAP logical and ‘&’.  

Return A LDAP expression

Parameters

• left: A LDAP expression.
• right: A LDAP expression.

LDAP logical or ‘|’.  

Return A LDAP expression

Parameters

• left: A LDAP expression.
• right: A LDAP expression.

cppmicroservices::LDAPPropExpr operator&& (const cppmicroservices::LDAPPropExpr & left, const cppmicroservices::LDAPPropExpr & right)

cppmicroservices::LDAPPropExpr operator|| (const cppmicroservices::LDAPPropExpr & left, const cppmicroservices::LDAPPropExpr & right)

class cppmicroservices::LDAPFilter
#include <cppmicroservices/LDAPFilter.h> An RFC 1960-based Filter.

A LDAPFilter can be used numerous times to determine if the match argument matches the filter string that was used to create the LDAPFilter.

Some examples of LDAP filters are:

• “(cn=Babs Jensen)”
• “!(cn=Tim Howes)”
• “(&(" + Constants::OBJECTCLASS + "=Person)(!((sn=Jensen)(cn=Babs J*))))”
• “(o=univ*of*mich*)”

Remark  This class is thread safe.

See  LDAPProp  for a fluent API generating LDAP filter strings

Public Functions

LDAPFilter()  
Creates a valid LDAPFilter object that matches nothing.

LDAPFilter(const std::string &filter)  
Creates a LDAPFilter object encapsulating the filter string.

This LDAPFilter object may be used to match a ServiceReference object or a ServiceProperties object.

If the filter cannot be parsed, an std::invalid_argument will be thrown with a human readable message where the filter became unparsable.

See  “Framework specification for a description of the filter string syntax.”  TODO!

Parameters
  • filter: The filter string.

Exceptions
  • std::invalid_argument: If filter contains an invalid filter string that cannot be parsed.

LDAPFilter(const LDAPFilter &other)  
~LDAPFilter()  
operator bool() const  

bool Match(const ServiceReferenceBase &reference) const  
Filter using a service’s properties.

This LDAPFilter is executed using the keys and values of the referenced service’s properties. The keys are looked up in a case insensitive manner.

Return  true if the service’s properties match this LDAPFilter false otherwise.

Parameters
  • reference: The reference to the service whose properties are used in the match.

bool Match(const Bundle &bundle) const  
Filter using a bundle’s manifest headers.

This LDAPFilter is executed using the keys and values of the bundle’s manifest headers. The keys are looked up in a case insensitive manner.

Return  true if the bundle’s manifest headers match this LDAPFilter false otherwise.

Parameters
  • bundle: The bundle whose manifest’s headers are used in the match.
Exceptions

- `std::runtime_error`: If the number of keys of the bundle’s manifest headers exceeds the value returned by `std::numeric_limits<int>::max()`.

bool `Match` (const `AnyMap` &`dictionary`) const

Filter using a `AnyMap` object with case insensitive key lookup.

This `LDAPFilter` is executed using the specified `AnyMap`'s keys and values. The keys are looked up in a case insensitive manner.

Return true if the `AnyMap`’s values match this filter; false otherwise.

Parameters

- `dictionary`: The `AnyMap` whose key/value pairs are used in the match.

Exceptions

- `std::runtime_error`: If the number of keys in the `dictionary` exceeds the value returned by `std::numeric_limits<int>::max()`.
- `std::runtime_error`: If the `dictionary` contains case variants of the same key name.

bool `MatchCase` (const `AnyMap` &`dictionary`) const

Filter using a `AnyMap`.

This `LDAPFilter` is executed using the specified `AnyMap`’s keys and values. The keys are looked up in a normal manner respecting case.

Return true if the `AnyMap`’s values match this filter; false otherwise.

Parameters

- `dictionary`: The `AnyMap` whose key/value pairs are used in the match.

Exceptions

- `std::runtime_error`: If the number of keys in the `dictionary` exceeds the value returned by `std::numeric_limits<int>::max()`.
- `std::runtime_error`: If the `dictionary` contains case variants of the same key name.

std::string `ToString` () const

Returns this `LDAPFilter`’s filter string.

The filter string is normalized by removing whitespace which does not affect the meaning of the filter.

Return This `LDAPFilter`'s filter string.

bool `operator==` (const `LDAPFilter` &`other`) const

Compares this `LDAPFilter` to another `LDAPFilter`.

This implementation returns the result of calling `this->ToString() == other.ToString()`.

Return Returns the result of calling `this->ToString() == other.ToString()`.

Parameters

- `other`: The object to compare against this `LDAPFilter`. 

LDAPFilter &operator=(const LDAPFilter &filter)

Protected Attributes

std::shared_ptr<LDAPFilterData> d

class cppmicroservices::LDAPProp
#include <cppmicroservices/LDAPProp.h> A fluent API for creating LDAP filter strings.
Examples for creating LDAPFilter objects:

// This creates the filter "(&(name=Ben)(!(count=1)))"
LDAPFilter filter(LDAPProp("name") == "Ben" && !(LDAPProp("count") == 1));

// This creates the filter "((presence=*)(!(absence=*))"
LDAPFilter filter(LDAPProp("presence") || !LDAPProp("absence"));

// This creates the filter "(&(ge>=-3)(approx~=hi))"
LDAPFilter filter(LDAPProp("ge") >= -3 && LDAPProp("approx").Approx("hi"));

See LDAPFilter

Public Functions

LDAPProp (std::string property)
Create a LDAPProp instance for the named LDAP property.

Parameters

• property: The name of the LDAP property.

LDAPPropExpr operator== (const std::string &s) const
LDAPPropExpr operator== (const cppmicroservices::Any &s) const
LDAPPropExpr operator== (bool b) const
template <class T>
LDAPPropExpr operator== (const T &s) const

operator LDAPPropExpr () const
States the absence of the LDAP property.

Return A LDAP expression object.

LDAPPropExpr operator!=(const std::string &s) const
LDAPPropExpr operator!=(const cppmicroservices::Any &s) const
template <class T>
LDAPPropExpr operator!= (const T &s) const
LDAPPropExpr operator>= (const std::string &s) const
LDAPPropExpr operator>= (const cppmicroservices::Any &s) const
template <class T>
LDAPPropExpr operator>= (const T & s) const

LDAPPropExpr operator<= (const std::string & s) const

LDAPPropExpr operator<= (const cppmicroservices::Any & s) const

template <class T>
LDAPPropExpr Approx (const std::string & s) const

LDAPPropExpr Approx (const cppmicroservices::Any & s) const

15.2.8 Listeners

using cppmicroservices::ServiceListener = typedef std::function<void(const ServiceEvent&)> A ServiceEvent listener.

A ServiceListener can be any callable object and is registered with the Framework using the BundleContext#AddServiceListener(const ServiceListener&, const std::string&) method. ServiceListener instances are called with a ServiceEvent object when a service has been registered, unregistered, or modified.

See ServiceEvent

using cppmicroservices::BundleListener = typedef std::function<void(const BundleEvent&)> A BundleEvent listener.

When a BundleEvent is fired, it is asynchronously (if threading support is enabled) delivered to a BundleListener. The Framework delivers BundleEvent objects to a BundleListener in order and does not concurrently call a BundleListener.

A BundleListener can be any callable object and is registered with the Framework using the BundleContext#AddBundleListener(const BundleListener&) method. BundleListener instances are called with a BundleEvent object when a bundle has been installed, resolved, started, stopped, updated, unresolved, or uninstalled.

See BundleEvent

using cppmicroservices::FrameworkListener = typedef std::function<void(const FrameworkEvent&)> A FrameworkEvent listener.

When a FrameworkEvent is fired, it is asynchronously (if threading support is enabled) delivered to a FrameworkListener. The Framework delivers FrameworkEvent objects to a FrameworkListener in order and does not concurrently call a FrameworkListener.

A FrameworkListener can be any callable object and is registered with the Framework using the BundleContext#AddFrameworkListener(const FrameworkListener&) method. FrameworkListener instances are called with a FrameworkEvent object when a framework life-cycle event or notification message occurred.

See FrameworkEvent

template <class R>
ServiceListener cppmicroservices::ServiceListenerMemberFunctor(R * receiver, void(R::*)(const ServiceEvent &) callback)
A convenience function that binds the member function callback of an object of type R and returns a ServiceListener object.
This object can then be passed into `AddServiceListener()`.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `std::bind` instead.

**Return** a `ServiceListener` object.

**Template Parameters**

- R: The type containing the member function.

**Parameters**

- `receiver`: The object of type R.
- `callback`: The member function pointer.

```cpp
template <class R>
BundleListener cppmicroservices::BundleListenerMemberFunctor(R * receiver, void(R::*)(const BundleEvent &callback)
```

A convenience function that binds the member function `callback` of an object of type R and returns a `BundleListener` object.

This object can then be passed into `AddBundleListener()`.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `std::bind` instead.

**Return** a `BundleListener` object.

**Template Parameters**

- R: The type containing the member function.

**Parameters**

- `receiver`: The object of type R.
- `callback`: The member function pointer.

```cpp
template <class R>
FrameworkListener cppmicroservices::BindFrameworkListenerToFunctor(R * receiver, void(R::*)(const FrameworkEvent &callback)
```

A convenience function that binds the member function `callback` of an object of type R and returns a `FrameworkListener` object.

This object can then be passed into `AddFrameworkListener()`.

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use `std::bind` instead.

**Return** a `FrameworkListener` object.

**Template Parameters**

- R: The type containing the member function.

**Parameters**

- `receiver`: The object of type R.
- `callback`: The member function pointer.
15.2.9 SharedLibrary

class cppmicroservices::SharedLibrary
The SharedLibrary class loads shared libraries at runtime.

Public Functions

SharedLibrary()

SharedLibrary(const SharedLibrary &other)

SharedLibrary(const std::string &libPath, const std::string &name)
Construct a SharedLibrary object using a library search path and a library base name.

Parameters

• libPath: An absolute path containing the shared library
• name: The base name of the shared library, without prefix and suffix.

SharedLibrary(const std::string &absoluteFilePath)
Construct a SharedLibrary object using an absolute file path to the shared library.
Using this constructor effectively disables all setters except SetFilePath().

Parameters

• absoluteFilePath: The absolute path to the shared library.

~SharedLibrary()
Destroy this object but does not unload the shared library.

SharedLibrary &operator=(const SharedLibrary &other)

void Load()
Loads the shared library pointed to by this SharedLibrary object.

On POSIX systems dlopen() is called with the RTLD_LAZY and RTLD_LOCAL flags unless the compiler is gcc 4.4.x or older. Then the RTLD_LAZY and RTLD_GLOBAL flags are used to load the shared library to work around RTTI problems across shared library boundaries.

Exceptions

• std::logic_error: If the library is already loaded.
• std::system_error: If loading the library failed.

void Load(int flags)
Loads the shared library pointed to by this SharedLibrary object, using the specified flags on POSIX systems.

Exceptions

• std::logic_error: If the library is already loaded.
• std::system_error: If loading the library failed.
void **Un**load ()
    Un-loads the shared library pointed to by this **SharedLibrary** object.

**Exceptions**

• std::**runtime_error**: If an error occurred while un-loading the shared library.

void **Set**Name **(const** std::string &**name**) **const**
    Sets the base name of the shared library.
    Does nothing if the shared library is already loaded or the **SharedLibrary(const std::string&)** constructor was used.

**Parameters**

• **name**: The base name of the shared library, without prefix and suffix.

std::string **Get**Name () **const**
    Gets the base name of the shared library.

Return  The shared libraries base name.

std::string **Get**FilePath (**const** std::string &**name**) **const**
    Gets the absolute file path for the shared library with base name **name**, using the search path returned by **GetLibraryPath()**.

Return  The absolute file path of the shared library.

**Parameters**

• **name**: The shared library base name.

void **Set**FilePath (**const** std::string &**absoluteFilePath**) **const**
    Sets the absolute file path of this **SharedLibrary** object.
    Using this methods with a non-empty **absoluteFilePath** argument effectively disables all other set-ters.

**Parameters**

• **absoluteFilePath**: The new absolute file path of this **SharedLibrary** object.

std::string **Get**FilePath () **const**
    Gets the absolute file path of this **SharedLibrary** object.

Return  The absolute file path of the shared library.

void **Set**LibraryPath (**const** std::string &**path**) **const**
    Sets a new library search path.
    Does nothing if the shared library is already loaded or the **SharedLibrary(const std::string&)** constructor was used.

**Parameters**

• **path**: The new shared library search path.
std::string GetLibraryPath() const
Gets the library search path of this SharedLibrary object.

**Return** The library search path.

void SetSuffix(const std::string &suffix)
Sets the suffix for shared library names (e.g. lib). Does nothing if the shared library is already loaded or the SharedLibrary(const std::string&) constructor was used.

**Parameters**
- suffix: The shared library name suffix.

std::string GetSuffix() const
Gets the file name suffix of this SharedLibrary object.

**Return** The file name suffix of the shared library.

void SetPrefix(const std::string &prefix)
Sets the file name prefix for shared library names (e.g. .dll or .so). Does nothing if the shared library is already loaded or the SharedLibrary(const std::string&) constructor was used.

**Parameters**
- prefix: The shared library name prefix.

std::string GetPrefix() const
Gets the file name prefix of this SharedLibrary object.

**Return** The file name prefix of the shared library.

void *GetHandle() const
Gets the internal handle of this SharedLibrary object.

**Return** nullptr if the shared library is not loaded, the operating system specific handle otherwise.

bool IsLoaded() const
Gets the loaded/unloaded stated of this SharedLibrary object.

**Return** true if the shared library is loaded, false otherwise.

### 15.2.10 ShrinkableMap

**template <class Key, class T>**
**class** cppmicroservices::ShrinkableMap
A std::map style associative container allowing query and removal operations only.
Public Types

template<typename container_type = std::map<Key, T>>
    using iterator = typename container_type::iterator

template<typename container_type = std::map<Key, T>>
    using const_iterator = typename container_type::const_iterator

template<typename container_type = std::map<Key, T>>
    using size_type = typename container_type::size_type

template<typename container_type = std::map<Key, T>>
    using key_type = typename container_type::key_type

template<typename container_type = std::map<Key, T>>
    using mapped_type = typename container_type::mapped_type

template<typename container_type = std::map<Key, T>>
    using value_type = typename container_type::value_type

template<typename container_type = std::map<Key, T>>
    using reference = typename container_type::reference

template<typename container_type = std::map<Key, T>>
    using const_reference = typename container_type::const_reference

Public Functions

ShrinkableMap()
iterator begin()
const_iterator begin() const
iterator end()
const_iterator end() const
void erase(iterator pos)
void erase(iterator first, iterator last)
size_type erase(const Key &key)
bool empty() const
void clear()
size_type size() const
size_type max_size() const
T &operator[](const Key &key)

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use at(size_type pos) instead.
T &at (const Key &key)
const T &at (const Key &key) const
size_type count (const Key &key) const
iterator find (const Key &key)
const_iterator find (const Key &key) const
std::pair<iterator, iterator> equal_range (const Key &key)
std::pair<const_iterator, const_iterator> equal_range (const Key &key) const
iterator lower_bound (const Key &key)
const_iterator lower_bound (const Key &key) const
iterator upper_bound (const Key &key)
const_iterator upper_bound (const Key &key) const

15.2.11 ShrinkableVector

template <class E>
class cppmicroservices::ShrinkableVector
A std::vector style container allowing query and removal operations only.

Public Types

template<>
using container_type = std::vector<E>
template<>
using iterator = typename container_type::iterator
template<>
using const_iterator = typename container_type::const_iterator
template<>
using size_type = typename container_type::size_type
template<>
using reference = typename container_type::reference
template<>
using const_reference = typename container_type::const_reference
template<>
using value_type = typename container_type::value_type

Public Functions

ShrinkableVector()
iterator begin ()
const_iterator begin () const
iterator end ()
const_iterator end() const
reference front ()
const_reference front() const
reference back ()
const_reference back() const
iterator erase (iterator pos)
iterator erase (iterator first, iterator last)
void pop_back ()
bool empty () const
void clear ()
size_type size () const
reference at (size_type pos)
const_reference at (size_type pos) const
const_reference operator[] (size_type i) const

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use at(size_type pos) instead.

reference operator[] (size_type i)

Deprecated since version 3.1.0: This function exists only to maintain backwards compatibility and will be removed in the next major release. Use at(size_type pos) instead.

15.3 Macros

Preprocessor macros provided by CppMicroServices.

15.3.1 CPPMICROSERVICES_INITIALIZE_BUNDLE

CPPMICROSERVICES_INITIALIZE_BUNDLE
Creates initialization code for a bundle.

Each bundle which wants to register itself with the CppMicroServices library has to put a call to this macro in one of its source files. Further, the bundle’s source files must be compiled with the US_BUNDLE_NAME pre-processor definition set to a bundle-unique identifier.

Calling the CPPMICROSERVICES_INITIALIZE_BUNDLE macro will initialize the bundle for use with the CppMicroServices library.

Hint: If you are using CMake, consider using the provided CMake macro usFunctionGenerateBundleInit().
15.3.2 CPPMICROSERVICES_INITIALIZE_STATIC_BUNDLE

CPPMICROSERVICES_INITIALIZE_STATIC_BUNDLE(_bundle_name)

Initialize a static bundle.

This macro initializes the static bundle named _bundle_name.

Parameters

- _bundle_name: The name of the bundle to initialize.

If the bundle provides an activator, use the CPPMICROSERVICES_IMPORT_BUNDLE macro instead, to ensure that the activator is referenced and can be called. Do not forget to actually link the static bundle to the importing executable or shared library.

See also:

CPPMICROSERVICES_IMPORT_BUNDLE
Static Bundles

15.3.3 CPPMICROSERVICES_IMPORT_BUNDLE

CPPMICROSERVICES_IMPORT_BUNDLE(_bundle_name)

Import a static bundle.

This macro imports the static bundle named _bundle_name.

Parameters

- _bundle_name: The name of the bundle to import.

Inserting this macro into your application’s source code will allow you to make use of a static bundle. It will initialize the static bundle and reference its BundleActivator. If the bundle does not provide an activator, use the CPPMICROSERVICES_INITIALIZE_STATIC_BUNDLE macro instead. Do not forget to actually link the static bundle to the importing executable or shared library.

Example:

```
#include "cppmicroservices/BundleImport.h"
CPPMICROSERVICES_IMPORT_BUNDLE(MyStaticBundle1)
```

See also:

CPPMICROSERVICES_INITIALIZE_STATIC_BUNDLE
Static Bundles

15.3.4 CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(_activator_type)

Export a bundle activator class.

Call this macro after the definition of your bundle activator to make it accessible by the CppMicroServices library.

Parameters
• _activator_type: The fully-qualified type-name of the bundle activator class.

Example:

```cpp
class MyActivator : public BundleActivator
{
public:
    void Start(BundleContext /*context*/)  
    { /* register stuff */ }

    void Stop(BundleContext /*context*/)  
    { /* cleanup */ }

};

CPPMICROSERVICES_EXPORT_BUNDLE_ACTIVATOR(MyActivator)
```

## 15.3.5 CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE

**CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE**(_service_interface_type, _service_interface_id)_

Declare a service interface id.

This macro associates the given identifier _service_interface_id_ (a string literal) to the interface class called _service_interface_type_. The Identifier must be unique. For example:

```cpp
#include "cppmicroservices/ServiceInterface.h"

struct ISomeInterface { ... };

CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE(ISomeInterface, "com.mycompany.service.ISomeInterface/1.0")
```

The usage of this macro is optional and the service interface id which is automatically associated with any type is usually good enough (the demangled type name). However, care must be taken if the default id is compared with a string literal hard-coding a service interface id. E.g. the default id for templated types in the STL may differ between platforms. For user-defined types and templates the ids are typically consistent, but platform specific default template arguments will lead to different ids.

This macro is normally used right after the class definition for _service_interface_type_, in a header file.

If you want to use **CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE** with interface classes declared in a namespace then you have to make sure the **CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE** macro call is not inside a namespace though. For example:

```cpp
#include "cppmicroservices/ServiceInterface.h"

namespace Foo
{
    struct ISomeInterface { ... };
}

CPPMICROSERVICES_DECLARE_SERVICE_INTERFACE(Foo::ISomeInterface, "com.mycompany.
└ service.ISomeInterface/1.0")
```

### Parameters

15.3. Macros 193
• _service_interface_type: The service interface type.
• _service_interface_id: A string literal representing a globally unique identifier.
16.1 BundleDTO

```cpp
struct cppmicroservices::framework::dto::BundleDTO
    #include <cppmicroservices/servicecomponent/runtime/dto/BundleDTO.hpp> A representation of a bundle.
```

**Public Members**

- `unsigned long id`
  The bundle’s unique identifier.

  **See** `Bundle::GetBundleId()`

- `unsigned long lastModified`
  The time when the bundle was last modified.

  **See** `Bundle::GetLastModified()`

- `uint32_t state`
  The bundle’s state.

  **See** `Bundle::GetState()`

- `std::string symbolicName`
  The bundle’s symbolic name.

  **See** `Bundle::GetSymbolicName()`

- `std::string version`
  The bundle’s version.
See `Bundle::GetVersion()`

## 16.2 ComponentConfigurationDTO

```cpp
enum gr_componentconfigurationdto::ComponentState
Values:

UNSATISFIED_REFERENCE
The component configuration is unsatisfied due to an unsatisfied reference.

SATISFIED
The component configuration is satisfied.

Any services declared by the component description are registered.

ACTIVE
The component configuration is active.

This is the normal operational state of a component configuration.
```

```cpp
struct cppmicroservices::service::component::runtime::dto::ComponentConfigurationDTO
#include <cppmicroservices/servicecomponent/runtime/dto/ComponentConfigurationDTO.hpp> A representa-
```

```cpp
tion of an actual instance of a declared component description parameterized by component properties.
```

### Public Members

- **ComponentDescriptionDTO** `description`
  The representation of the component configuration’s component description.

- **ComponentState** `state`
  The current state of the component configuration.

  This is one of `ComponentState::UNSATISFIED_REFERENCE`, `ComponentState::SATISFIED` or `ComponentState::ACTIVE`.

- **unsigned long** `id`
  The id of the component configuration.

  The id is a non-persistent, unique value assigned at runtime. The id is also available as the `component.id` component property. The value of this field is unspecified if the state of this component configuration is unsatisfied.

- **std::unordered_map<std::string, cppmicroservices::Any>** `properties`
  The component properties for the component configuration.

  See `ComponentContext::GetProperties()`

- **std::vector<SatisfiedReferenceDTO>** `satisfiedReferences`
  The satisfied references.

  Each `SatisfiedReferenceDTO` in the vector represents a satisfied reference of the component configuration. The vector must be empty if the component configuration has no satisfied references.

- **std::vector<UnsatisfiedReferenceDTO>** `unsatisfiedReferences`
  The unsatisfied references.

  Each `UnsatisfiedReferenceDTO` in the vector represents an unsatisfied reference of the component configuration. The vector must be empty if the component configuration has no unsatisfied references.
16.3 ComponentConstants

**const std::string cppmicroservices::service::component::ComponentConstants::SERVICE_COMPONENT**

Manifest key specifying the Service Component descriptions.

The attribute value may be retrieved from the `cppmicroservices::AnyMap` object returned by the `Bundle::GetHeaders` method.

**const std::string cppmicroservices::service::component::ComponentConstants::COMPONENT_NAME**

A component property for a component configuration that contains the name of the component as specified in the `name` attribute of the `component` element.

The value of this property must be of type `std::string`.

**const std::string cppmicroservices::service::component::ComponentConstants::COMPONENT_ID**

A component property that contains the generated id for a component configuration.

The value of this property must be of type `unsigned long`.

The value of this property is assigned by Service Component Runtime when a component configuration is created. Service Component Runtime assigns a unique value that is larger than all previously assigned values since Service Component Runtime was started. These values are NOT persistent across restarts of Service Component Runtime.

**US_ServiceComponent_EXPORT const std::string cppmicroservices::service::component::ComponentConstants::COMPONENT_FACTORY**

A service registration property for a Component Factory that contains the value of the `factory` attribute. The value of this property must be of type `std::string`.

**US_ServiceComponent_EXPORT const std::string cppmicroservices::service::component::ComponentConstants::REFERENCE_TARGET_SUFFIX**

The suffix for reference target properties. These properties contain the filter to select the target services for a reference. The value of this property must be of type `std::string`.

**US_ServiceComponent_EXPORT const std::string cppmicroservices::service::component::ComponentConstants::REFERENCE_SCOPE_PROTOTYPE_REQUIRED**

Reference scope is `prototype_required`.

The reference is satisfied only if the service is registered with `PROTOTYPE` scope. Each component instance receives a distinct service object.

**US_ServiceComponent_EXPORT const std::string cppmicroservices::service::component::ComponentConstants::CONFIG_POLICY_IGNORE**

Constants used for ComponentMetadata configurationPolicy attribute.

16.4 ComponentContext

**class cppmicroservices::service::component::ComponentContext**

`#include <cppmicroservices/servicecomponent/ComponentContext.hpp>` A Component Context object is used by a component instance to interact with its execution context including locating services by reference name.

Each component instance has a unique Component Context.
A component instance may obtain its Component Context object through its activate, modified, and deactivate methods.

**Public Functions**

```cpp
virtual ~ComponentContext ()

virtual std::unordered_map<std::string, cppmicroservices::Any> GetProperties () const = 0
    Returns the component properties for this Component Context.

Return The properties for this Component Context.
```

```cpp
virtual cppmicroservices::BundleContext GetBundleContext () const = 0
    Returns the BundleContext of the bundle which contains this component.

Return The BundleContext of the bundle containing this component.
```

```cpp
virtual cppmicroservices::Bundle GetUsingBundle () const = 0
    If the component instance is registered as a service using the servicescope="bundle" or servicescope="prototype" attribute, then this method returns the bundle using the service provided by the component instance.

This method will return an invalid Bundle if:

- The component instance is not a service, then no bundle can be using it as a service.
- The component instance is a service but did not specify the servicescope="bundle" or servicescope="prototype" attribute, then all bundles using the service provided by the component instance will share the same component instance.
- The service provided by the component instance is not currently being used by any bundle.

Return The bundle using the component instance as a service or an invalid bundle.
```

```cpp
virtual void EnableComponent (const std::string &name) = 0
    Enables the specified component name.

The specified component name must be in the same bundle as this component.

This method must return after changing the enabled state of the specified component name. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

Parameters

- name: The name of a component or empty string to indicate all components in the bundle.
```

```cpp
virtual void DisableComponent (const std::string &name) = 0
    Disables the specified component name.

The specified component name must be in the same bundle as this component.

This method must return after changing the enabled state of the specified component name. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

Parameters

```
virtual cppmicroservices::ServiceReferenceBase GetServiceReference() const = 0

If the component instance is registered as a service using the service element, then this method returns
the service reference of the service provided by this component instance.

This method will return an invalid ServiceReference object if the component instance is not registered as a
service.

Return The ServiceReference object for the component instance or invalid object if the component in-
stance is not registered as a service.

template <class T>
std::shared_ptr<T> LocateService(const std::string &refName) const

Returns the service object for the specified reference name and type.

If the cardinality of the reference is 0..n or 1..n and multiple services are bound to the reference, the
service with the highest ranking (as specified in its Constants::SERVICE_RANKING property) is returned.
If there is a tie in ranking, the service with the lowest service id (as specified in its Constants::SERVICE_ID
property); that is, the service that was registered first is returned.

Return A service object for the referenced service or nullptr if the reference cardinality is 0..1 or
0..n and no bound service is available.

Parameters

• refName: The name of a reference as specified in a reference element in this component’s
description.

Exceptions

• ComponentException: If Service Component Runtime catches an exception while activating
the bound service.

template <class T>
std::vector<std::shared_ptr<T>> LocateServices(const std::string &refName) const

Returns the service objects for the specified reference name and type.

Return A vector of service objects for the referenced service or empty vector if the reference cardinality
is 0..1 or 0..n and no bound service is available. If the reference cardinality is 0..1 or 1..1 and
a bound service is available, the vector will have exactly one element.

Parameters

• refName: The name of a reference as specified in a reference element in this component’s
description.

Exceptions

• ComponentException: If Service Component Runtime catches an exception while activating
a bound service.

Protected Functions

virtual std::shared_ptr<void> LocateService(const std::string &name, const std::string &type) const = 0

Returns the service object for the specified reference name and type.
If the cardinality of the reference is 0..n or 1..n and multiple services are bound to the reference, the service with the highest ranking (as specified in its Constants::SERVICE_RANKING property) is returned. If there is a tie in ranking, the service with the lowest service id (as specified in its Constants::SERVICE_ID property); that is, the service that was registered first is returned.

**Return** A service object for the referenced service or nullptr if the reference cardinality is 0..1 or 0..n and no bound service is available.

**Parameters**
- **name**: The name of a reference as specified in a reference element in this component's description.
- **type**: The service interface type.

**Exceptions**
- **ComponentException**: If Service Component Runtime catches an exception while activating the bound service.

```cpp
virtual std::vector<std::shared_ptr<void>> LocateServices(const std::string &name, const std::string &type) const = 0
```

Returns the service objects for the specified reference name and type.

**Return** A vector of service objects for the referenced service or empty vector if the reference cardinality is 0..1 or 0..n and no bound service is available. If the reference cardinality is 0..1 or 1..1 and a bound service is available, the vector will have exactly one element.

**Parameters**
- **name**: The name of a reference as specified in a reference element in this component's description.
- **type**: The service interface type.

**Exceptions**
- **ComponentException**: If Service Component Runtime catches an exception while activating a bound service.

### 16.5 ComponentDescriptionDTO

```cpp
struct ComponentDescriptionDTO
```

A representation of a declared component description.

**Public Members**

- **name**: The name of the component.
  - This is declared in the name attribute of the component element. This must be the default name if the component description does not declare a name.
  ```cpp
ccppmicroservices::framework::dto::BundleDTO bundle
```
  - The bundle declaring the component description.
std::string **scope**
The service scope.

This is declared in the **scope** attribute of the **service** element. This must be empty string if the component description does not declare any service interfaces.

std::string **implementationClass**
The fully qualified name of the implementation class.

This is declared in the **implementation-class** element.

bool **defaultEnabled**
The initial enabled state.

This is declared in the **enabled** attribute of the **component** element.

bool **immediate**
The immediate state.

This is declared in the **immediate** attribute of the **component** element.

std::vector<std::string> **serviceInterfaces**
The fully qualified names of the service interfaces.

These are declared in the **interface** attribute of the **provide** elements. The vector must be empty if the component description does not declare any service interfaces.

std::unordered_map<std::string, cppmicroservices::Any> **properties**
The declared component properties.

These are declared in the **property** and **properties** elements.

std::vector<ReferenceDTO> **references**
The referenced services.

These are declared in the **reference** elements. The vector must be empty if the component description does not declare references to any services.

std::string **activate**
The name of the activate method.

This is declared in the **activate** attribute of the **component** element. This must be the default value which is “Activate”, if the component description does not declare an activate method name.

std::string **deactivate**
The name of the deactivate method.

This is declared in the **deactivate** attribute of the **component** element. This must be the default value which is “Deactivate”, if the component description does not declare a deactivate method name.

std::string **modified**
The name of the modified method.

This is declared in the **modified** attribute of the **component** element. This must be the default value which is “Modified”, if the component description does not declare a modified method name.

### 16.6 ComponentException

**class** cppmicroservices::service::component::ComponentException

```
#include <cppmicroservices/servicecomponent/ComponentException.hpp> Exception which may be thrown by Service Component Runtime.
```
Inherits from std::runtime_error

**Public Functions**

ComponentException (const std::string &message)
Construct a new ComponentException with the specified message.

Parameters

- message: The message for the exception.

ComponentException (const char *message)
virtual ~ComponentException()

16.7 ReferenceDTO

struct cppmicroservices::service::component::runtime::dto::ReferenceDTO
#include <cppmicroservices/servicecomponent/runtime/dto/ReferenceDTO.hpp> A representation of a declared reference to a service.

**Public Members**

std::string name
The name of the reference.
This is declared in the name attribute of the reference element. This must be the default name if the component description does not declare a name for the reference.

std::string interfaceName
The service interface of the reference.
This is declared in the interface attribute of the reference element.

std::string cardinality
The cardinality of the reference.
This is declared in the cardinality attribute of the reference element. This must be the default cardinality if the component description does not declare a cardinality for the reference.

std::string policy
The policy of the reference.
This is declared in the policy attribute of the reference element. This must be the default policy if the component description does not declare a policy for the reference.

std::string policyOption
The policy option of the reference.
This is declared in the policy-option attribute of the reference element. This must be the default policy option if the component description does not declare a policy option for the reference.

std::string target
The target of the reference.
This is declared in the `target` attribute of the `reference` element. This must be an empty string if the component description does not declare a target for the reference.

```cpp
std::string bind
```

The name of the bind method of the reference.

This is declared in the `bind` attribute of the `reference` element. This must be an empty string if the component description does not declare a bind method for the reference.

```cpp
std::string unbind
```

The name of the unbind method of the reference.

This is declared in the `unbind` attribute of the `reference` element. This must be an empty string if the component description does not declare an unbind method for the reference.

```cpp
std::string updated
```

The name of the updated method of the reference.

This is declared in the `updated` attribute of the `reference` element. This must be an empty string if the component description does not declare an updated method for the reference.

```cpp
std::string scope
```

The scope of the reference.

This is declared in the `scope` attribute of the `reference` element. This must be the default scope if the component description does not declare a scope for the reference.

### 16.8 SatisfiedReferenceDTO

```cpp
struct cppmicroservices::service::component::runtime::dto::SatisfiedReferenceDTO
```

```cpp
#include <cppmicroservices/servicecomponent/runtime/dto/SatisfiedReferenceDTO.hpp> A representation of a satisfied reference.
```

#### Public Members

```cpp
std::string name
```

The name of the declared reference.

This is declared in the `name` attribute of the `reference` element of the component description.

See `ReferenceDTO::name`

```cpp
std::string target
```

The target property of the satisfied reference.

This is the value of the `component property` whose name is the concatenation of the `declared reference name` and ".target". This must be empty string if no target property is set for the reference.

```cpp
std::vector<cppmicroservices::framework::dto::ServiceReferenceDTO> boundServices
```

The bound services.

Each `ServiceReferenceDTO` in the vector represents a service bound to the satisfied reference. The vector must be empty if there are no bound services.
16.9 ServiceComponentRuntime

class cppmicroservices::service::component::runtime::ServiceComponentRuntime

#include <cppmicroservices/servicecomponent/runtime/ServiceComponentRuntime.hpp> The ServiceComponentRuntime service represents the Declarative Services actor, known as Service Component Runtime (SCR), that manages the service components and their life cycle.

The ServiceComponentRuntime service allows introspection of the components managed by Service Component Runtime.

This service differentiates between a ComponentDescriptionDTO and a ComponentConfigurationDTO. A ComponentDescriptionDTO is a representation of a declared component description. A ComponentConfigurationDTO is a representation of an actual instance of a declared component description parameterized by component properties.

Public Functions

virtual ~ServiceComponentRuntime()

virtual std::vector<dto::ComponentDescriptionDTO> GetComponentDescriptionDTOs(const std::vector<cppmicroservices::Bundle>& bundles = {}, const = 0)

Returns the component descriptions declared by the specified active bundles.

Only component descriptions from active bundles are returned. If the specified bundles have no declared components or are not active, an empty collection is returned.

Return The declared component descriptions of the specified active bundles. An empty collection is returned if there are no component descriptions for the specified active bundles.

Parameters

• bundles: The bundles whose declared component descriptions are to be returned. Specifying no bundles, or the equivalent of an empty Bundle vector, will return the declared component descriptions from all active bundles.

virtual dto::ComponentDescriptionDTO GetComponentDescriptionDTO(const cppmicroservices::Bundle &bundle, const std::string &name) const = 0

Returns the ComponentDescriptionDTO declared with the specified name by the specified bundle.

Only component descriptions from active bundles are returned. An empty object is returned if no such component is declared by the given bundle or the bundle is not active.

Return The declared component description or empty object if the specified bundle is not active or does not declare a component description with the specified name.

Parameters

• bundle: The bundle declaring the component description.

• name: The name of the component description.
virtual std::vector<dto::ComponentConfigurationDTO> GetComponentConfigurationDTOs (const
dto::ComponentDescription &description) const = 0

Returns the component configurations for the specified component description.

**Return** A vector containing a snapshot of the current component configurations for the specified compo-
nent description. An empty vector is returned if there are none.

**Parameters**

- **description**: The component description.

---

virtual bool IsComponentEnabled (const dto::ComponentDescription &description) const = 0

Returns whether the specified component description is currently enabled.

The enabled state of a component description is initially set by the *enabled* attribute of the component
description.

**Return** true if the specified component description is currently enabled. Otherwise, false.

**See** EnableComponent

**See** DisableComponent

**See** ComponentContext::DisableComponent(std::string)

**See** ComponentContext::EnableComponent(std::string)

**Parameters**

- **description**: The component description.

---

virtual std::shared_future<void> EnableComponent (const dto::ComponentDescriptionDTO &description) = 0

Enables the specified component description.

If the specified component description is currently enabled, this method has no effect.

This method must return after changing the enabled state of the specified component description. *Any* actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

**Remark** The shared_future<void> returned can contain a { cppmicroservices::SecurityException if en-
abling the component caused the bundle’s shared library to be loaded and the bundle failed a security
check.

**Return** A future that will be ready when the actions that result from changing the enabled state of the
specified component have completed.

**See** IsComponentEnabled

**Parameters**

- **description**: The component description to enable.
virtual std::shared_future<void> DisableComponent (const dto::ComponentDescriptionDTO &description) = 0

Disables the specified component description.
If the specified component description is currently disabled, this method has no effect.
This method must return after changing the enabled state of the specified component description. Any actions that result from this, such as activating or deactivating a component configuration, must occur asynchronously to this method call.

Return A future that will be ready when the actions that result from changing the enabled state of the specified component have completed.

See IsComponentEnabled

Parameters
  • description: The component description to disable.

16.10 ServiceReferenceDTO

struct cppmicroservices::framework::dto::ServiceReferenceDTO
#include <cppmicroservices/servicecomponent/runtime/dto/ServiceReferenceDTO.hpp> A representation of a satisfied reference.

Public Members

unsigned long id
  The id of the service.

See Constants::SERVICE_ID

unsigned long bundle
  The id of the bundle that registered the service.

See ServiceReference::GetBundle()

std::unordered_map<std::string, cppmicroservices::Any> properties
  The properties for the service.
  The value type must be a numerical type, Boolean, String or a container of any of the former.

See ServiceReference::GetProperty(String)

std::vector<unsigned long> usingBundles
  The ids of the bundles that are using the service.

See ServiceReference::GetUsingBundles()
16.11 UnsatisfiedReferenceDTO

```
struct cppmicroservices::service::component::runtime::dto::UnsatisfiedReferenceDTO
#include <cppmicroservices/servicecomponent/runtime/dto/UnsatisfiedReferenceDTO.hpp> A representation
of an unsatisfied reference.

Public Members

std::string name
The name of the declared reference.

This is declared in the name attribute of the reference element of the component description.

See ReferenceDTO::name

std::string target
The target property of the unsatisfied reference.

This is the value of the component property whose name is the concatenation of the declared reference
name and ”.target”. This must be empty string if no target property is set for the reference.

std::vector<cppmicroservices::framework::dto::ServiceReferenceDTO> targetServices
The target services.

Each ServiceReferenceDTO in the vector represents a target service for the reference. The vector must be
empty if there are no target services. The upper bound on the number of target services in the vector is the
upper bound on the cardinality of the reference.
17.1 Configuration

class cppmicroservices::service::cm::Configuration
#include <cppmicroservices/cm/Configuration.hpp> The Configuration object (normally obtained as a std::shared_ptr<Configuration>) is the principal means for clients of ConfigurationAdmin to inspect or update the Configuration of a given service or service factory.

Public Functions

virtual ~Configuration ()

virtual std::string GetPid () const = 0
Get the PID of this Configuration.

Return the PID of this Configuration

Exceptions
• std::runtime_error: if this Configuration object has been Removed

virtual std::string GetFactoryPid () const = 0
Get the Factory PID which is responsible for this Configuration.

If this Configuration does not belong to any Factory, returns an empty string.

Return the Factory PID associated with this Configuration, if applicable

Exceptions
• std::runtime_error: if this Configuration object has been Removed
virtual AnyMap GetProperties () const = 0
Get the properties of this Configuration.
Returns a copy.

Return the properties of this Configuration

Exceptions
• std::runtime_error: if this Configuration object has been Removed

virtual unsigned long GetChangeCount () const = 0
Get the change count.
Each Configuration must maintain a change counter that is incremented with a positive value every time
the configuration is updated and its properties are stored. The counter must be incremented before the
targets are updated and events are sent out.

Return A monotonically increasing value reflecting changes in this Configuration.

Exceptions
• std::runtime_error: if this Configuration object has been Removed

virtual std::shared_future<void> Update (AnyMap properties = AnyMap {
    AnyMap::UNORDERED_MAP_CASEINSENSITIVE_KEYS}
) = 0
Update the properties of this Configuration.
Invoking this method will trigger the ConfigurationAdmin impl to push the updated properties to any
ManagedService / ManagedServiceFactory / ConfigurationListener which has a matching PID / Factory
PID.
If the properties are empty, the Configuration will not be removed, but instead updated with an empty
properties map.

Remark The shared_future<void> returned can contain a cppmicroservices::SecurityException if the
Configuration caused a bundle’s shared library to be loaded and the bundle failed a security check.

Return a shared_future<void> which can be used to wait for the asynchronous operation that pushed the
update to a ManagedService, ManagedServiceFactory or ConfigurationListener to complete. If an
exception occurs during the execution of the service component’s Modified method, this exception is
intercepted and logged by Declarative Services. This exception is not returned in the shared_future.

Exceptions
• std::runtime_error: if this Configuration object has been Removed

Parameters
• properties: The properties to update this Configuration with.

virtual std::pair<bool, std::shared_future<void>> UpdateIfDifferent (AnyMap properties = AnyMap {
    AnyMap::UNORDERED_MAP_CASEINSENSITIVE_KEYS}
) = 0
Update the properties of this Configuration if they differ from the current properties.
Invoking this method will trigger the ConfigurationAdmin impl to push the updated properties to any
ManagedService / ManagedServiceFactory / ConfigurationListener which has a matching PID / Factory
PID, but only if the properties differ from the current properties. It will return true in this case, and false otherwise.

If the properties are empty, the Configuration will not be removed, but instead updated with an empty properties map, unless it already had empty properties.

**Remark** The shared_future<void> returned can contain a cppmicroservices::SecurityException if the Configuration caused a bundle’s shared library to be loaded and the bundle failed a security check.

**Return** std::pair<boolean, std::shared_future<void>> The boolean indicates whether the properties were updated or not. The shared_future<void> allows access to the result of the asynchronous operation that pushed the update operation to a ManagedService, ManagedServiceFactory or ConfigurationListener. If an exception occurs during the execution of the service component’s Modified method, this exception is intercepted and logged by Declarative Services. This exception is not returned in the shared_future.

**Exceptions**

- std::runtime_error: if this Configuration object has been Removed

**Parameters**

- properties: The properties to update this Configuration with (if they differ)

**virtual** std::shared_future<void> Remove () = 0

Remove this Configuration from ConfigurationAdmin.

This will trigger a push to any ConfigurationListener. This will also trigger the ConfigurationAdmin implementation to update any corresponding ManagedService with an empty AnyMap. Any corresponding ManagedServiceFactory will have its Removed method invoked with the corresponding PID.

**Return** a shared_future<void> to access the result of the asynchronous operation that pushed the remove operation to a ManagedService, ManagedServiceFactory or ConfigurationListener. If an exception occurs during the execution of the service component’s Modified method, this exception is intercepted and logged by Declarative Services. This exception is not returned in the shared_future.

**Exceptions**

- std::runtime_error: if this Configuration object has been Removed already

### 17.2 ConfigurationAdmin

**class** cppmicroservices::service::cm::ConfigurationAdmin

#include <cppmicroservices/cm/ConfigurationAdmin.hpp> The ConfigurationAdmin interface is the means by which applications and services can interact with the Configuration objects at runtime.

It can be used to create or obtain Configuration objects, which can in turn be queried for their properties or be used to update the properties of a given service or service factory.

**Public Functions**

**virtual ~ConfigurationAdmin()**

**virtual** std::shared_ptr<Configuration> GetConfiguration (const std::string &pid) = 0

Get an existing or new Configuration object.
If the *Configuration* object for this PID does not exist, create a new *Configuration* object for that PID with empty properties.

**Return** the *Configuration* object for this PID

**Parameters**

- `pid`: The PID to get the *Configuration* for

```cpp
virtual std::shared_ptr<Configuration> CreateFactoryConfiguration(const std::string &factoryPid) = 0
```

Create a new *Configuration* object for a *ManagedServiceFactory*.

The factoryPid is the PID of the *ManagedServiceFactory* (which must be different from the PIDs of any services it manages) and the instanceName will be a randomly-generated, unique name. The *Configuration* object’s properties are empty. The returned *Configuration* will have a PID of the form $factoryPid~$instanceName.

**Return** a new *Configuration* object for this Factory PID with a randomly-generated unique name

**Parameters**

- `factoryPid`: The Factory PID to create a new *Configuration* for

```cpp
virtual std::shared_ptr<Configuration> GetFactoryConfiguration(const std::string &factoryPid, const std::string &instanceName) = 0
```

Get an existing or new *Configuration* object for a *ManagedServiceFactory*.

The factoryPid is the PID of the *ManagedServiceFactory* (which must be different from the PIDs of any services it manages) and the instanceName is the unique name of one of those managed services. If the *Configuration* object for this combination of factoryPid and instanceName does not exist, create a new *Configuration* object for that combination, where properties are empty. The returned *Configuration* will have a PID of the form $factoryPid~$instanceName.

**Return** the *Configuration* object for this combination of Factory PID and instance name

**Parameters**

- `factoryPid`: The Factory PID to use to get an existing *Configuration* or create a new *Configuration* for
- `instanceName`: The unique name of an instance of a service managed by the *ManagedServiceFactory*

```cpp
virtual std::vector<std::shared_ptr<Configuration>> ListConfigurations(const std::string &filter = {}) = 0
```

Used to list all of the available configurations.

An LDAP filter expression can be used to filter based on any property of the configuration, including service.pid and service.factoryPid

**Return** a vector of *Configurations* matching the filter

**Parameters**

- `filter`: An optional filter expression to limit the Configurations which are returned, or empty for all.
17.3 ConfigurationException

class cppmicroservices::service::cm::ConfigurationException
#include <cppmicroservices/cm/ConfigurationException.hpp> Exception which may be thrown by Managed-
Service or ManagedServiceFactory subclasses to indicate to the ConfigurationAdmin implementation that the
Configuration they have been given is invalid.

The ConfigurationAdmin implementation will log the exception with as much detail as it can. The Configuration-
Exception class is not final to ensure it can be used with std::throw_with_nested - the ConfigurationAdmin
implementation will attempt to print the details of any nested exceptions as well.

Inherits from std::runtime_error

Public Functions

ConfigurationException (std::string rsn, std::string prop = "")
Construct a newConfigurationException with the specified reason and optionally specify which property
caused the error.

Parameters

• rsn: The reason for the exception.
• prop: The property which caused the exception, if applicable.

std::string GetReason () const
Returns the reason for this exception.

Return The reason for this exception.

std::string GetProperty () const
Returns the property which was responsible for this exception being throws, if applicable.

Could be empty.

Return The property which caused this exception.

virtual ~ConfigurationException()

17.4 ConfigurationListener

enum gr_configurationlistener::ConfigurationEventType
Values:

CM_UPDATED = 1
The ConfigurationEvent type for when a Configuration object has been updated.

CM_DELETED = 2
The ConfigurationEvent type for when a Configuration object has been removed.

class cppmicroservices::service::cm::ConfigurationEvent
#include <cppmicroservices/cm/ConfigurationListener.hpp> The ConfigurationEvent object is passed to the
ConfigurationListener when the configuration object for any service is updated or removed by ConfigurationAd-
min.
Public Functions

`ConfigurationEvent(const ServiceReference<ConfigurationAdmin> configAdmin, const ConfigurationEventType type, const std::string factoryPid, const std::string pid)`

`const ServiceReference<ConfigurationAdmin> &getReference() const`  
Get the `ServiceReference` object of the `ConfigurationAdmin` Service that created this event.

`Return` the service reference of this `ConfigurationEvent`

`const std::string &getPid() const`  
Get the PID of this `ConfigurationEvent`.  

`Return` the PID of this `ConfigurationEvent`

`const std::string &getFactoryPid() const`  
Get the Factory PID which is responsible for this `Configuration`.  

`Return` the Factory PID of this `ConfigurationEvent`

`ConfigurationEventType getType() const`  
Get the type of this `Configuration`.  

`Return` the `ConfigurationEventType` of this `ConfigurationEvent`

class cppmicroservices::service::cm::ConfigurationListener  
#include <cppmicroservices/cm/ConfigurationListener.hpp> The `ConfigurationListener` interface is the interface that Declarative Services implements to receive updates from the `ConfigurationAdmin` implementation for all configuration object updates.

`ConfigurationAdmin` sends updates for all configuration object updates. It is up to DS to determine if this is a configuration object associated with a service being managed by DS.

Public Functions

`virtual void configurationEvent (const ConfigurationEvent &event) = 0`  
Called whenever the `Configuration` for any service is updated or removed from `ConfigurationAdmin`.  

`Remark` This class is threadsafe  

`Parameters`

- `ConfigurationEvent`: object containing the `ConfigurationAdmin` service reference of the `ConfigurationAdmin` service that updated the configuration object, the PID or FactoryPid for the configuration object and the type of the update operation (update or remove)

`virtual ~ConfigurationListener()`
17.5 ManagedService

class cppmicroservices::service::cm::ManagedService
#include <cppmicroservices/cm/ManagedService.hpp> The ManagedService interface is the interface that services should implement to receive updates from the ConfigurationAdmin implementation with their Configuration.

The Configuration will be provided based on the service.pid ServiceProperty of the Service being registered with the CppMicroServices Framework. If this isn’t provided, the component.name property injected by DeclarativeServices will be used instead. If no Configuration is available for either of these, an empty AnyMap will be used to call Updated. Per the OSGi Spec, ManagedService is intended to be used for services which have singleton scope, where there will only be one implementation of a given interface.

Public Functions

virtual ~ManagedService() = 0

virtual void Updated(const AnyMap &properties) = 0
Called whenever the Configuration for this service is updated or removed from ConfigurationAdmin, and when the ManagedService is first registered with the Framework, to provide the initial Configuration.

Can throw aConfigurationException if there’s a problem with the properties. This exception will be logged by the ConfigurationAdminImpl to aid the application author’s investigation into the incorrect configuration.

Will be called asynchronously from the Service’s registration or an update to the Configuration.

Exceptions

• ConfigurationException: if something is wrong with the properties provided.

Parameters

• properties: The properties from the new/initial Configuration for this ManagedService

17.6 ManagedServiceFactory

class cppmicroservices::service::cm::ManagedServiceFactory
#include <cppmicroservices/cm/ManagedServiceFactory.hpp> The ManagedServiceFactory interface is the interface that service factories should implement to receive updates from the ConfigurationAdmin implementation with the Configurations for the services that the Factory will provide.

The service.factoryPid ServiceProperty of the ManagedServiceFactory (or the component.name which is injected by DeclarativeServices if no service.factoryPid is provided) will be used to select all Configurations applicable to this factory. Per the OSGi Spec, the ManagedServiceFactory is intended to be used when multiple instances of a given Service will exist in the Framework, but each with different Configurations. For this reason, ManagedServiceFactory implementations are encouraged to mirror any properties (excluding security critical properties) into the ServiceProperty map when publishing the service with the Framework (if it is going to be published). Clients of the service can then filter on the properties they require.

The Updated method will be invoked once for each initial Configuration for this factory and then again whenever any of those Configurations change or whenever any new Configurations for this factory are added.

The Removed method will be invoked whenever any Configuration which this factory has previously been configured with is removed from the ConfigurationAdmin implementation.
Public Functions

virtual ~ManagedServiceFactory()  

virtual void Updated (const std::string &pid, const AnyMap &properties) = 0  
Called whenever any Configuration for this service factory is updated with ConfigurationAdmin, and (possibly multiple times) when the ManagedServiceFactory is first registered with the Framework.

Can throw a ConfigurationException if there’s a problem with the properties. This exception will be logged by the ConfigurationAdminImpl to aid the application author’s investigation into the incorrect configuration.

Will be called asynchronously from the Service registration or an update to the Configuration.

The ManagedServiceFactory should update the corresponding service instance with the properties provided, and potentially update the properties of that service’s registration with the Framework (if it is registered).

If a corresponding service instance does not exist for this pid, the ManagedServiceFactory should create one with the properties provided. It should also register that new service with the Framework, if applicable.

Exceptions
  • ConfigurationException: if something is wrong with the properties provided.

Parameters
  • pid: The unique pid for this Configuration, of the form "$FACTORY_PID~$INSTANCE_NAME"
  • properties: The properties for this Configuration

virtual void Removed (const std::string &pid) = 0  
Called whenever one of the Configurations for this service is removed from ConfigurationAdmin.

Will be called asynchronously from the removal of the Configuration.

The ManagedServiceFactory should remove the corresponding service instance, and if it is registered with the Framework, it should be unregistered.

Parameters
  • pid: The unique pid for the Configuration to remove, of the form "$FACTORY_PID~$INSTANCE_NAME"
18.1 LogService

enum gr_logservice::SeverityLevel
Values:

LOG_ERROR = 1
Indicates the bundle or service may not be functional. Action should be taken to correct this situation.

LOG_WARNING = 2
Indicates a bundle or service is still functioning but may experience problems in the future because of the
warning condition.

LOG_INFO = 3
May be the result of any change in the bundle or service and does not indicate a problem.

LOG_DEBUG = 4
Used for problem determination and may be irrelevant to anyone but the bundle developer.

class cppmicroservices::logservice::LogService
#include <cppmicroservices/logservice/LogService.hpp> Provides methods for bundles to write messages to the
log.

LogService methods are provided to log messages; optionally with a ServiceReference object or an exception.
Bundles must log messages in the OSGi environment with a severity level according to the following hierarchy:

1. LOG_ERROR
2. LOG_WARNING
3. LOG_INFO
4. LOG_DEBUG

Remark This class is thread safe.
Public Functions

virtual ~LogService()

virtual void Log(SeverityLevel level, const std::string &message) = 0
Logs a message.

Parameters

• level: The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
• message: Human readable string describing the condition or empty string.

virtual void Log(SeverityLevel level, const std::string &message, const std::exception_ptr ex) = 0
Logs a message.

Parameters

• level: The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
• message: Human readable string describing the condition or empty string.
• ex: The exception that reflects the condition or nullptr.

virtual void Log(const ServiceReferenceBase &sr, SeverityLevel level, const std::string &message) = 0
Logs a message.

Parameters

• sr: The ServiceReferenceBase object of the service that this message is associated with or an invalid object.
• level: The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
• message: Human readable string describing the condition or empty string.

virtual void Log(const ServiceReferenceBase &sr, SeverityLevel level, const std::string &message, const std::exception_ptr ex) = 0
Logs a message with an exception associated and a ServiceReference object.

Parameters

• sr: The ServiceReferenceBase object of the service that this message is associated with or an invalid object.
• level: The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
• message: Human readable string describing the condition or empty string.
• ex: The exception that reflects the condition or nullptr.
19.1 AsyncWorkService

class cppmicroservices::async::AsyncWorkService
#include <cppmicroservices/asyncworkservice/AsyncWorkService.hpp> Provides a method which controls how DeclarativeServices internally schedules asynchronous work.

Creating an AsyncWorkService implementation is not required; this is intended to be used in specialty situations where the client application has requirements that the default asynchronous work scheduling mechanism does not conform to.

Remark This class is thread safe.

Public Functions

virtual ~AsyncWorkService()  

virtual void post (std::packaged_task<void>() > & task = 0Run a std::packaged_task<void()> (optionally on another thread asynchronously).

The std::future<void> associated with the std::packaged_task<void()> task object will contain the result from the task object.

Note The caller is required to manage the std::future<void> associated with the std::packaged_task<void()> in order to wait on the async task.

Parameters

- task: A std::packaged_task<void()> wrapping a Callable target to execute asynchronously.
These classes are the main API to the HttpService bundle

**Warning:** The Http Service API is not final and may be incomplete. It also may change between minor releases without backwards compatibility guarantees.

## 20.1 HttpServlet

**class** cppmicroservices::HttpServlet  
Inherits from std::enable_shared_from_this< HttpServlet >  
Subclassed by cppmicroservices::AbstractWebConsolePlugin

### Public Functions

**HttpServlet ()**

**virtual void Init (const ServletConfig &config)**  
Called by the servlet container to indicate to a servlet that the servlet is being placed into service.

The servlet container calls the Init method exactly once after instantiating the servlet. The Init method must complete successfully before the servlet can receive any requests.

The servlet container cannot place the servlet into service if the Init method

1. Throws a ServletException
2. Does not return within a time period defined by the Web server

**See** UnavailableException

**See** GetServletConfig
Parameters

- **config**: a ServletConfig object containing the servlet’s configuration and initialization parameters

Exceptions

- **ServletException**: if an exception has occurred that interferes with the servlet’s normal operation

```cpp
virtual void Destroy()
```

Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.

This method is only called once all threads within the servlet’s `service` method have exited or after a timeout period has passed. After the servlet container calls this method, it will not call the `service` method again on this servlet.

This method gives the servlet an opportunity to clean up any resources that are being held (for example, memory, file handles, threads) and make sure that any persistent state is synchronized with the servlet’s current state in memory.

```cpp
ServletConfig GetServletConfig() const
```

Returns a ServletConfig object, which contains initialization and startup parameters for this servlet.

The ServletConfig object returned is the one passed to the `Init` method.

**Return** the ServletConfig object that initializes this servlet

See `Init`

```cpp
virtual void Service(HttpServletRequest &request, HttpServletResponse &response)
```

```cpp
std::shared_ptr<ServletContext> GetServletContext() const
```

```cpp
virtual ~HttpServlet()
```

#### Public Static Attributes

```cpp
const std::string PROP_CONTEXT_ROOT
```

#### Protected Functions

```cpp
virtual long long GetLastModified(HttpServletRequest &request)
```

```cpp
virtual void DoGet(HttpServletRequest &request, HttpServletResponse &response)
```

```cpp
virtual void DoHead(HttpServletRequest &request, HttpServletResponse &response)
```

```cpp
virtual void DoDelete(HttpServletRequest &request, HttpServletResponse &response)
```

```cpp
virtual void DoPost(HttpServletRequest &request, HttpServletResponse &response)
```

```cpp
virtual void DoPut(HttpServletRequest &request, HttpServletResponse &response)
```

```cpp
virtual void DoTrace(HttpServletRequest &request, HttpServletResponse &response)
```

```cpp
std::unique_lock<std::mutex> Lock() const
```
### 20.2 HttpServletRequest

**Public Functions**

```cpp
~HttpServletRequest ()
HttpServletRequest (const HttpServletRequest &o)
HttpServletRequest &operator= (const HttpServletRequest &o)
std::shared_ptr<ServletContext> GetServletContext () const
Any GetAttribute (const std::string &name) const
std::vector<std::string> GetAttributeNames () const
std::size_t GetContentLength () const
std::string GetContentType () const
std::string GetLocalName () const
std::string GetRemoteHost () const
int GetLocalPort () const
int GetRemotePort () const
std::string GetScheme () const
std::string GetServerName () const
int GetServerPort () const
std::string GetProtocol () const
std::string GetContextPath () const
std::string GetPathInfo () const
std::string GetRequestUri () const
std::string GetRequestUrl () const
std::string GetServletPath () const
std::string GetQueryString () const
std::string GetHeader (const std::string &name) const
long long GetDateHeader (const std::string &name) const
std::vector<std::string> GetHeaderNames () const
std::vector<std::string> GetHeaders (const std::string &name) const
std::string GetMethod () const
```
std::vector<std::pair<std::string, float>> GetAcceptHeader() const
void RemoveAttribute(const std::string &name)
void SetAttribute(const std::string &name, const Any &value)

20.3 HttpServletResponse

class cppmicroservices::HttpServletResponse

Public Functions

virtual ~HttpServletResponse() override
HttpServletResponse(const HttpServletResponse &o)
HttpServletResponse &operator=(const HttpServletResponse &o)
void FlushBuffer()
bool IsCommitted() const
std::size_t GetBufferSize() const
std::string GetCharacterEncoding() const
std::string GetContentType() const
std::ostream &GetOutputStream()
void Reset()
void ResetBuffer()
void SetBufferSize(std::size_t size)
void SetCharacterEncoding(const std::string &charset)
void SetContentLength(std::size_t size)
void SetContentType(const std::string &type)
void AddHeader(const std::string &name, const std::string &value)
void SetHeader(const std::string &name, const std::string &value)
void SetDateHeader(const std::string &name, long long date)
void AddIntHeader(const std::string &name, int value)
void SetIntHeader(const std::string &name, int value)
bool ContainsHeader(const std::string &name) const
std::string GetHeader(const std::string &name) const
int GetStatus() const
void `SetStatus` (int `statusCode`)
void `SendError` (int `statusCode`, `const std::string &msg` = `std::string()`)  
void `SendRedirect` (`const std::string &location`)

**Public Static Attributes**

`const int SC_CONTINUE`  
Status code (100) indicating the client can continue.

`const int SC_SWITCHING_PROTOCOLS`  
Status code (101) indicating the server is switching protocols according to Upgrade header.

`const int SC_OK`  
Status code (200) indicating the request succeeded normally.

`const int SC_CREATED`  
Status code (201) indicating the request succeeded and created a new resource on the server.

`const int SC_ACCEPTED`  
Status code (202) indicating that a request was accepted for processing, but was not completed.

`const int SC_NON_AUTHORITATIVE_INFORMATION`  
Status code (203) indicating that the meta information presented by the client did not originate from the server.

`const int SC_NO_CONTENT`  
Status code (204) indicating that the request succeeded but that there was no new information to return.

`const int SC_RESET_CONTENT`  
Status code (205) indicating that the agent SHOULD reset the document view which caused the request to be sent.

`const int SC_PARTIAL_CONTENT`  
Status code (206) indicating that the server has fulfilled the partial GET request for the resource.

`const int SC_MULTIPLE_CHOICES`  
Status code (300) indicating that the requested resource corresponds to any one of a set of representations, each with its own specific location.

`const int SC_MOVED_PERMANENTLY`  
Status code (301) indicating that the resource has permanently moved to a new location, and that future references should use a new URI with their requests.

`const int SC_FOUND`  
Status code (302) indicating that the resource reside temporarily under a different URI.

`const int SC_MOVED_TEMPORARILY`  
Status code (302) indicating that the resource has temporarily moved to another location, but that future references should still use the original URI to access the resource.

`const int SC_SEE_OTHER`  
Status code (303) indicating that the response to the request can be found under a different URI.

`const int SC_NOT_MODIFIED`  
Status code (304) indicating that a conditional GET operation found that the resource was available and not modified.
const int SC_USE_PROXY
  Status code (305) indicating that the requested resource MUST be accessed through the proxy given by
  the Location field.

const int SC_TEMPORARY_REDIRECT
  Status code (307) indicating that the requested resource resides temporarily under a different URI.

const int SC_BAD_REQUEST
  Status code (400) indicating the request sent by the client was syntactically incorrect.

const int SC_UNAUTHORIZED
  Status code (401) indicating that the request requires HTTP authentication.

const int SC_PAYMENT_REQUIRED
  Status code (402) reserved for future use.

const int SC_FORBIDDEN
  Status code (403) indicating the server understood the request but refused to fulfill it.

const int SC_NOT_FOUND
  Status code (404) indicating that the requested resource is not available.

const int SC_METHOD_NOT_ALLOWED
  Status code (405) indicating that the method specified in the Request-Line is not allowed for the resource
  identified by the Request-URI.

const int SC_NOT_ACCEPTABLE
  Status code (406) indicating that the resource identified by the request is only capable of generating re-
  sponse entities which have content characteristics not acceptable according to the accept headers sent in
  the request.

const int SC_PROXY_AUTHENTICATION_REQUIRED
  Status code (407) indicating that the client MUST first authenticate itself with the proxy.

const int SC_REQUEST_TIMEOUT
  Status code (408) indicating that the client did not produce a request within the time that the server was
  prepared to wait.

const int SC_CONFLICT
  Status code (409) indicating that the request could not be completed due to a conflict with the current state
  of the resource.

const int SC_GONE
  Status code (410) indicating that the resource is no longer available at the server and no forwarding address
  is known.

const int SC_LENGTH_REQUIRED
  Status code (411) indicating that the request cannot be handled without a defined Content-Length.

const int SC_PRECONDITION_FAILED
  Status code (412) indicating that the precondition given in one or more of the request-header fields evalu-
  ated to false when it was tested on the server.

const int SC_REQUEST_ENTITY_TOO_LARGE
  Status code (413) indicating that the server is refusing to process the request because the request entity is
  larger than the server is willing or able to process.

const int SC_REQUEST_URI_TOO_LONG
  Status code (414) indicating that the server is refusing to service the request because the Request-URI is
  longer than the server is willing to interpret.
const int SC_UNSUPPORTED_MEDIA_TYPE
Status code (415) indicating that the server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

const int SC_REQUESTED_RANGE_NOT_SATISFIABLE
Status code (416) indicating that the server cannot serve the requested byte range.

const int SC_EXPECTATION_FAILED
Status code (417) indicating that the server could not meet the expectation given in the Expect request header.

const int SC_INTERNAL_SERVER_ERROR
Status code (500) indicating an error inside the HTTP server which prevented it from fulfilling the request.

const int SC_NOT_IMPLEMENTED
Status code (501) indicating the HTTP server does not support the functionality needed to fulfill the request.

const int SC_BAD_GATEWAY
Status code (502) indicating that the HTTP server received an invalid response from a server it consulted when acting as a proxy or gateway.

const int SC_SERVICE_UNAVAILABLE
Status code (503) indicating that the HTTP server is temporarily overloaded, and unable to handle the request.

const int SC_GATEWAY_TIMEOUT
Status code (504) indicating that the server did not receive a timely response from the upstream server while acting as a gateway or proxy.

const int SC_HTTP_VERSION_NOT_SUPPORTED
Status code (505) indicating that the server does not support or refuses to support the HTTP protocol version that was used in the request message.

Protected Functions

HttpServletResponse (HttpServletResponsePrivate *d)

virtual std::streambuf *GetOutputStreamBuffer ()

void SetOutputStreamBuffer (std::streambuf *sb)

Protected Attributes

HttpServletResponsePrivate *d

20.4 ServletConfig

class cppmicroservices::ServletConfig
A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
**Public Functions**

`ServletConfig()`

`ServletConfig(const ServletConfig &other)`

`virtual ~ServletConfig()`

`std::string GetServletName(const) const` — Returns the name of this servlet instance.

The name may be provided via server administration, assigned in the web application deployment descriptor, or for an unregistered (and thus unnamed) servlet instance it will be the servlet’s class name.

`std::shared_ptr<ServletContext> GetServletContext(const) const` — Returns a reference to the `ServletContext` in which the caller is executing.

**Protected Functions**

`void SetServletName(const std::string &name)`

`void SetServletContext(const std::shared_ptr<ServletContext> &context)`

## 20.5 ServletContainer

class `cppmicroservices::ServletContainer`

**Public Functions**

`ServletContainer(BundleContext bundleCtx, const std::string &contextPath = std::string())`

`~ServletContainer()`

`void SetContextPath(const std::string &contextPath)`

`std::string GetContextPath(const) const`

`void Start()`

`void Stop()`

`std::shared_ptr<ServletContext> GetContext(const std::string &uripath) const`

`std::string GetContextPath(const ServletContext *context) const`
### 20.6 ServletContext

**class** cppmicroservices::ServletContext

**Public Functions**

`std::string GetContextPath () const`

`std::shared_ptr<ServletContext> GetContext (const std::string &uripath)`

`std::string GetMimeType (const std::string &file) const`
CHAPTER 21

Web Console API

These classes are the main API to the WebConsole bundle

**Warning:** The Web Console API is not final and may be incomplete. It also may change between minor releases without backwards compatibility guarantees.

### 21.1 AbstractWebConsolePlugin

**class** `cppmicroservices::AbstractWebConsolePlugin`

The Web Console can be extended by registering an OSGi service for the interface `HttpServlet` with the service property `org.cppmicroservices.webconsole.label` set to the label (last segment in the URL) of the page.

The respective service is called a Web Console Plugin or a plugin for short.

To help rendering the response the Web Console bundle provides two options. One of the options is to extend the `AbstractWebConsolePlugin` overwriting the `RenderContent(HttpServletRequest&, HttpServletResponse&)` method.

Inherits from `cppmicroservices::HttpServlet`

Subclassed by `cppmicroservices::SimpleWebConsolePlugin`

**Public Types**

```cpp
using TemplateData = Kainjow::Mustache::Data
```
**Public Functions**

```cpp
virtual std::string GetLabel () const = 0
Retrieves the label.
This is the last component in the servlet path.

Return the label.
```

```cpp
virtual std::string GetTitle () const = 0
Retrieves the title of the plug-in.
It is displayed in the page header and is also included in the title of the HTML document.

Return the plugin title.
```

```cpp
virtual std::string GetCategory () const
This method should return category string which will be used to render the plugin in the navigation menu.
Default implementation returns null, which will result in the plugin link rendered as top level menu item.
Concrete implementations wishing to be rendered as a sub-menu item under a category should override this method and return a string or define org.cppmicroservices.webconsole.category service property. Currently only single level categories are supported. So, this should be a simple string.

Return category
```

```cpp
virtual std::shared_ptr<WebConsoleVariableResolver> GetVariableResolver (HttpServletRequest &request)
```

```cpp
virtual void SetVariableResolver (HttpServletRequest &request, const std::shared_ptr<WebConsoleVariableResolver> &resolver)
```

**Protected Functions**

```cpp
virtual bool IsHtmlRequest (HttpServletRequest &request)
Detects whether this request is intended to have the headers and footers of this plugin be rendered or not.
This method always returns true and may be overwritten by plugins to detect from the actual request, whether or not to render the header and footer.

Return true if the page should have headers and footers rendered

Parameters
• request: the original request passed from the HTTP server
```

```cpp
virtual void DoGet (HttpServletRequest &request, HttpServletResponse &response)
Renders the web console page for the request.
This consist of the following five parts called in order:
1. Send back a requested resource
2. StartResponse(HttpServletRequest&, HttpServletResponse&)
3. RenderTopNavigation(HttpServletRequest&, std::ostream&)
```
4. `RenderContent(HttpServletRequest&, HttpServletResponse&)`
5. `EndResponse(HttpServletRequest&, std::ostream&)`

**Note:** If a resource is sent back for the request only the first step is executed. Otherwise the first step is a null-operation actually and the latter four steps are executed in order.

If the `IsHtmlRequest(HttpServletRequest&)` method returns `false` only the `RenderContent(HttpServletRequest&, HttpServletResponse&)` method is called.

See `HttpServlet::DoGet(HttpServletRequest&, HttpServletResponse&)`

```cpp
virtual void RenderContent (HttpServletRequest &request, HttpServletResponse &response) = 0
```
This method is used to render the content of the plug-in.

It is called internally from the Web Console.

**Parameters**
- `request`: the HTTP request send from the user
- `response`: the HTTP response object, where to render the plugin data.

```cpp
std::ostream &StartResponse (HttpServletRequest &request, HttpServletResponse &response)
```
This method is responsible for generating the top heading of the page.

**Return** the stream that was used for generating the response.

See `EndResponse(HttpServletRequest&, std::ostream&)`

**Parameters**
- `request`: the HTTP request coming from the user
- `response`: the HTTP response, where data is rendered

```cpp
void RenderTopNavigation (HttpServletRequest &request, std::ostream &writer)
```
This method is called to generate the top level links with the available plug-ins.

**Parameters**
- `request`: the HTTP request coming from the user
- `writer`: the writer, where the HTML data is rendered

```cpp
void EndResponse (HttpServletRequest &request, std::ostream &writer)
```
This method is responsible for generating the footer of the page.

See `StartResponse(HttpServletRequest&, HttpServletResponse&)`

**Parameters**
- `request`: the HTTP request coming from the user
- `writer`: the writer, where the HTML data is rendered
std::vector<std::string> GetCssReferences() const
Returns a list of CSS reference paths or null if no additional CSS files are provided by the plugin.

The result is an array of strings which are used as the value of the href attribute of the <link> elements placed in the head section of the HTML generated. If the reference is a relative path, it is turned into an absolute path by prepending the value of the WebConsoleConstants::ATTR_APP_ROOT request attribute.

Return The list of additional CSS files to reference in the head section or an empty list if no such CSS files are required.

std::string ReadTemplateFile(const std::string &templateFile, cppmicroservices::BundleContext context = cppmicroservices::GetBundleContext()) const

21.2 SimpleWebConsolePlugin

class cppmicroservices::SimpleWebConsolePlugin
SimpleWebConsolePlugin is a utility class that provides a default implementation of the AbstractWebConsolePlugin and supports the following features:

• Methods for (un)registering the web console plugin service
• Default implementation for resource loading

Inherits from cppmicroservices::AbstractWebConsolePlugin

Public Functions

SimpleWebConsolePlugin(const std::string &label, const std::string &title, std::string category = std::string(), std::vector<std::string> css = std::vector<std::string>())
Creates new Simple Web Console Plugin with the given category.

Parameters

• label: the front label. See AbstractWebConsolePlugin::GetLabel()
• title: the plugin title. See AbstractWebConsolePlugin::GetTitle()
• category: the plugin's navigation category. See AbstractWebConsolePlugin::GetCategory()
• css: the additional plugin CSS. See AbstractWebConsolePlugin::GetCssReferences()

std::string GetLabel() const
See AbstractWebConsolePlugin::GetLabel()

std::string GetTitle() const
See AbstractWebConsolePlugin::GetTitle()

std::string GetCategory() const
See AbstractWebConsolePlugin::GetCategory()
```cpp
std::shared_ptr<SimpleWebConsolePlugin> Register(const BundleContext &context = GetBundleContext())

This is an utility method.

It is used to register the plugin service. Don’t forget to call Unregister() when the plugin is no longer needed.

**Return** A shared pointer to this plugin.

**Parameters**

- `context`: the bundle context used for service registration.

```cpp
void Unregister()

An utility method that removes the service, registered by the Register(const BundleContext&) method.

**Protected Functions**

```cpp
std::vector<std::string> GetCssReferences() const

See AbstractWebConsolePlugin::GetCssReferences()

BundleContext GetContext() const
```

### 21.3 WebConsoleConstants

```cpp
struct cppmicroservices::WebConsoleConstants

WebConsoleConstants provides some common constants that are used by plugin developers.

**Public Static Attributes**

```cpp
std::string SERVICE_NAME

The name of the service to register as to be used as a “plugin” for the web console (value is “cppmicroservices::HttpServlet”).

std::string PLUGIN_LABEL

The URI address label under which the Web Console plugin is called (value is “org.cppmicroservices.webconsole.label”).

This service registration property must be set to a single non-empty string value. Otherwise the Servlet services will be ignored by the Web Console and not be used as a plugin.

std::string PLUGIN_TITLE

The title under which the Web Console plugin is called (value is “org.cppmicroservices.webconsole.title”).

For Servlet services not extending the AbstractWebConsolePlugin this property is required for the service to be used as a plugin. Otherwise the service is just ignored by the Web Console.

For Servlet services extending from the AbstractWebConsolePlugin abstract class this property is not technically required. To support lazy service access, e.g. for plugins implemented using the Service Factory pattern, the use of this service registration property is encouraged.

std::string PLUGIN_CATEGORY

The category under which the Web Console plugin is listed in the top navigation (value is “org.cppmicroservices.webconsole.category”).
```
For *Servlet* services not extending the *AbstractWebConsolePlugin* this property is required to declare a specific category. Otherwise the plugin is put into the default category.

For *Servlet* services extending from the *AbstractWebConsolePlugin* abstract class this property is not technically required. To support lazy service access with categorization, e.g. for plugins implemented using the *Service Factory* pattern, the use of this service registration property is strongly encouraged. If the property is missing the *AbstractWebConsolePlugin::GetCategory()* is called which should be overwritten.

```cpp
std::string PLUGIN_CSS_REFERENCES
```

The name of the service registration properties providing references to addition CSS files that should be loaded when rendering the header for a registered plugin.

This property is expected to be a single string value or a vector of string values.

This service registration property is only used for plugins registered as *SERVICE_NAME* services which do not extend the *AbstractWebConsolePlugin*. Extensions of the *AbstractWebConsolePlugin* should overwrite the *AbstractWebConsolePlugin::GetCssReferences()* method to provide additional CSS resources.

```cpp
std::string ATTR_APP_ROOT
```

The name of the request attribute providing the absolute path of the Web Console root (value is “org.cppmicroservices.webconsole.appRoot”).

This consists of the servlet context path (from *HttpServletRequest::GetContextPath()* and the Web Console servlet path (from *HttpServletRequest::GetServletPath()* /us/console by default).

The type of this request attribute is `std::string`.

```cpp
std::string ATTR_PLUGIN_ROOT
```

The name of the request attribute providing the absolute path of the current plugin (value is “org.cppmicroservices.webconsole.pluginRoot”).

This consists of the servlet context path (from *HttpServletRequest::GetContextPath()*), the configured path of the web console root (/us/console by default) and the plugin label *PLUGIN_LABEL*.

The type of this request attribute is `std::string`.

```cpp
std::string ATTR_LABEL_MAP
```

The name of the request attribute providing a mapping of labels to page titles of registered console plugins (value is “org.cppmicroservices.webconsole.labelMap”).

This map may be used to render a navigation of the console plugins as the *AbstractWebConsolePlugin::RenderTopNavigation(HttpServletRequest&, std::ostream&)* method does.

The type of this request attribute is `AnyMap`.

```cpp
std::string ATTR_CONSOLE_VARIABLE_RESOLVER
```

The name of the request attribute holding the *WebConsoleVariableResolver* for the request (value is “org.cppmicroservices.webconsole.variable.resolver”).

See *WebConsoleVariableResolver*

See *AbstractWebConsolePlugin::GetVariableResolver*

See *AbstractWebConsolePlugin::SetVariableResolver*
21.4 WebConsoleDefaultVariableResolver

class cppmicroservices::WebConsoleDefaultVariableResolver
The default Web Console variable resolver class.
This variable resolver uses Mustache template logic to resolve variables.
Inherits from cppmicroservices::WebConsoleVariableResolver

Public Functions

virtual std::string Resolve (const std::string &variable) const

Returns a replacement value for the named variable.

Return The replacement value.

Parameters

• variable: The name of the variable for which to return a replacement.

MustacheData &GetData ()

21.5 WebConsoleVariableResolver

struct cppmicroservices::WebConsoleVariableResolver
The WebConsoleVariableResolver interface defines the API for an object which may be provided by plugins to provide replacement values for variables in the generated content.

Plugins should call the AbstractWebConsolePlugin::SetVariableResolver method to provide their implementation for variable resolution.

The main use of such a variable resolver is when a plugin is using a static template which provides slots to place dynamically generated content parts.

Note: The variable resolver must be set in the request before the response stream is retrieved calling the HttpServletResponse::GetOutputStream method. Otherwise the variable resolver will not be used for resolving variables.

See AbstractWebConsolePlugin::GetVariableResolver(HttpServletRequest&)
See AbstractWebConsolePlugin::SetVariableResolver

Subclassed by cppmicroservices::WebConsoleDefaultVariableResolver

Public Functions

virtual ~WebConsoleVariableResolver ()

virtual std::string Resolve (const std::string &variable) const = 0

Returns a replacement value for the named variable.
**Return** The replacement value.

**Parameters**
- `variable`: The name of the variable for which to return a replacement.
These classes are the main API to the Shell Service bundle

**Warning:** The Shell Service API is not final and may be incomplete. It also may change between minor releases without backwards compatibility guarantees.

### 22.1 ShellService

```cpp
class cppmicroservices::ShellService
```

**Public Functions**

```cpp
ShellService() 
~ShellService() 
void ExecuteCommand(const std::string &cmd)
std::vector<std::string> GetCompletions(const std::string &in)
```
All notable changes to this project will be documented in this file. The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

23.1 (UNRELEASED) vX.X.X (20XX-XX-XX)

Full Changelog

23.2 Added

23.3 Changed

23.4 Removed

23.5 Deprecated

23.6 Fixed

23.7 v3.7.2 (2022-06-16)

Full Changelog
23.8 General Note

The last time CppMicroServices was upgraded to a new version on GitHub was two years ago. We think it will not be useful to include every change since then; below we have captured all of the relevant changes between v3.6.0 and v3.7.2.

23.9 Added

- [Log Service] LogService Implementation
- [Declarative Services] Added thread pool to DS
- [Core Framework] Anymap erase and compare
- Provide updated manifest.json schema in repo
- [Configuration Admin/Declarative Services] Configadmin ds integration
- GitHub Actions used as CI/CD solution
- [AsyncWorkService] Added AsyncWorkService interface to CppMicroServices
- GitHub Actions runs workflow for clang build with ASAN, TSAN, and UBSAN enabled
- [AsyncWorkService] AsyncWorkService DS Integration
- [AsyncWorkService] AsyncWorkService ConfigAdmin Integration
- [Security/Core Framework] bundle validation mechanism

23.10 Changed

- [Core Framework] Migrated all test driver tests to gtest
- Optimize the string creation in us_service_interface_iid<void>()
- [Core Framework] Improve the performance of removing service listeners
- [Declarative Services] Reduce possibility for symbol conflicts with autogenerated DS files
- Switch project to c++17
- Upgraded to CMake 3.17
- [Core Framework] Switch code to use std::string_view instead of abseil
- [Core Framework] Integrate LogService core framework and add more detail to exception messages

23.11 Removed

- [Core Framework] Remove Bundle Threads
- TravisCI and Appveyor removed as CI/CD solution
23.12 Deprecated

23.13 Fixed

- [Core Framework] Fix ServiceTracker race
- [Core Framework] Made ToJSON for strings do proper escaping of special characters
- [Core Framework] Fix issues with line endings in BundleResourceTest.cpp test
- Fix #301 so TSAN builds work on Linux
- [Configuration Admin] ConfigAdmin remove notification when configuration object is created
- [Configuration Admin] Fix TestConcurrentBindUnbind error
- [Core Framework] Fix ServiceTracker race
- [Configuration Admin] RemoveBoundServicesCache fix
- [Declarative Services] Prevent nullptr from being passed to service constructors
- [Declarative Services] fix segfault when service class ctor throws
- [Core Framework] Fix bug when an AddingService method returns nullptr
- [Configuration Admin] Fix deadlock in ConfigAdmin Update, UpdateIfDifferent, and Remove
- [Configuration Admin] Fix ConfigurationAdmin ListConfigurations and Fix Race Conditions in DS
- [Resource Compiler] Allow ResourceCompiler to accept Unicode Command-line arguments
- [Core Framework] service tracker doesn’t track services after a close and then open
- Fix warning suppression leakage to non-CppMicroServices code inside translation units
- [Core Framework] Fix leak in ServiceListeners.cpp
- [Core Framework] Ensure that any BundleContext functions do not segfault if the bundle context is invalid
- Fixed issue where DS/CA do not build if US_BUILD_TESTING isn’t ON
- Fixed build issues when building in “getting_started” dir
- [Core Framework] Fix check-then-act-race in GetServiceFromFactory
- [Core Framework] Fix bug in FindResources() for data-only bundles
- [Core Framework] Fix check-then-act race for BundleContext
- [Declarative Services] Fix inheritance for ComponentException
- [Core Framework] Add –max-threads flag for core framework mem test
- [Core Framework] Fix leaked file handle

23.14 v3.6.0 (2020-08-13)

Full Changelog
23.15 **Added**

- [Declarative Services] Support dynamic policy reference option
- Added initial implementation of Configuration Admin

23.16 **Changed**

- BundleContext::InstallBundles

23.17 **Removed**

23.18 **Deprecated**

23.19 **Fixed**

- Fixed data race in BundleRegistry::Install
- Fixed race condition in Declarative Services
- Removed gtest dependency when not building the tests

23.20 **v3.5.0 (2020-07-04)**

Full Changelog

23.21 **Added**

- Bundle::GetSymbol API
- SharedLibraryException

23.22 **Changed**

- Migrate a handful of tests from the legacy test suite to gtest based test suite
- Improve shared library loading error messages
23.23 Removed

- Remove dead code and partially implemented features
- Remove code with license conflicts

23.24 Deprecated

23.25 Fixed

- Correctly install Declarative Services and LogService headers
- Infinite loop in GetCurrentWorkingDir
- Use cross build objcopy
- Service reference dependency deadlock
- Instantiating multiple service implementations within the same service component
- Codecov integration
- BundleRegistry deadlock
- Remove unnecessary copying of AnyMap
- Minimum and maximum cardinality values
- Error if duplicate service component reference names are used
- Improve performance of ServiceTrackers

23.26 v3.4.0 (2019-12-10)

Full Changelog

23.27 Added

- Declarative Services
- Expose checksum from zip archive.
- Framework property (org.cppmicroservices.library.load.options) to control library loading options on macOS and Linux.
- Add gmock
23.28 Changed

23.29 Removed

23.30 Deprecated

• The following Bundle method functions:
  – GetProperties
  – GetProperty
  – GetPropertyKeys

23.31 Fixed

• static ServiceTracker object crashes in ServiceTracker::Close()
• Does the ServiceTracker deleter close the service?
• Optimize peak heap allocation when installing bundles
• Change GetHeaders API to return a const ref
• How do service consumers know whether to use BundleContext::GetService or ServiceObjects?
• Add a testpoint to validate the return value of ServiceFactory::GetService
• Invalid Bundle causes crash on method invocation
• Use correct framework event severity and exception types for service factory errors
• Raspberry Pi arm build failing
• Service ctor exception crash
• Update library loading error messages
• Unknown Cmake Command “add_compile_definitions”
• GetChildResources() should not have a dependency on GetChildren()
• Improved code coverage to 90%
• Various performance improvements to:
  – Reduce the number of open file handles
  – Reduce peak heap memory utilization
  – AtCompoundKey
  – ServiceTracker
  – Service look up
  – Bundle installs
23.32 v3.3.0 (2018-02-20)

Full Changelog

23.33 Added

- Support constructing long LDAP expressions using concise C++ #246
- Bundle manifest validation #182

23.34 Fixed

- Fix seg faults when using default constructed LDAPFilter #251

23.35 v3.2.0 (2017-10-30)

Full Changelog

23.36 Added

- Code coverage metrics. #219
- GTest integration. #200
- Support boolean properties in LDAP filter creation. #224
- Unicode support. #245

23.37 Changed

- Re-enable single-threaded build configuration. #239

23.38 Fixed

- Fix a race condition when getting and ungetting a service. #202
- Make reading the current working directory thread-safe. #209
- Guard against recursive service factory calls. #213
- Fix LDAP filter match logic to properly handle keys starting with the same sub-string. #227
- Fix seg fault when using a default constructed LDAPFilter instance. #232
- Several fixes with respect to error code handling. #238
- IsConvertibleTo method doesn’t check for validity of member. #240
23.39 v3.1.0 (2017-06-01)

Full Changelog

23.39.1 Changed

• Improved BadAnyCastException message. #181
• Support installing bundles that do not have .DLL/.so/.dylib file extensions. #205

23.39.2 Deprecated

• The following BundleContext member functions:
  – RemoveBundleListener
  – RemoveFrameworkListener
  – RemoveServiceListener
And the variants of
  – AddBundleListener
  – AddFrameworkListener,
  – AddServiceListener
that take member functions.
• The free functions:
  – ServiceListenerMemberFunctor
  – BundleListenerMemberFunctor
  – BindFrameworkListenerToFunctor
• The functions
  – ShrinkableVector::operator[std::size_t]
  – ShrinkableMap::operator[const Key&]

23.39.3 Fixed

• Cannot add more than one listener if its expressed as a lambda. #95
• Removing Listeners does not work well #83
• Crash when trying to acquire bundle context #172
• Fix for unsafe_any_cast #198
• Stopping a framework while bundle threads are still running may deadlock #210
23.40 v3.0.0 (2017-02-08)

Full Changelog

See the migration guide for moving from a 2.x release to 3.x.

23.40.1 Added

- Added MinGW-w64 to the continuous integration matrix #168
- Include major version number in library names and install dirs #144
- Integrated coverity scan reports #16
- Added OS X to the continuous integration matrix #136
- Building for Android is now supported #106
- Enhanced the project structure to support sub-projects #14
- The bundle life-cycle now supports all states as described by OSGi and is controllable by the user #25
- Added support for framework listeners and improved logging #40
- Implemented framework properties #42
- Static bundles embedded into an executable are now auto-installed #109
- LDAP queries can now be run against bundle meta-data #53
- Resources from bundles can now be accessed without loading their shared library #15
- Support last modified time for embedded resources #13

23.40.2 Changed

- Fix up bundle property and manifest header handling #135
- Introduced C++11 features #35
- Re-organize header files #43, #67
- Improved memory management for framework objects and services #38
- Removed static globals #31
- Switched to using OSGi nomenclature in class names and functions #46
- Improved static bundle support #21
- The resource compiler was ported to C++ and gained improved command line options #55
- Changed System Bundle ID to 0 #45
- Output exception details (if available) for troubleshooting #27
- Using the US_DECLARE_SERVICE_INTERFACE macro is now optional #24
- The Any::ToString() function now outputs JSON formatted text #12

23.40.3 Removed

- The autoload feature was removed from the framework #75
23.40.4 Fixed

- Headers with _p.h suffix do not get resolved in Xcode for automatic-tracking of counterparts #93
- usUtils.cpp - Crash can occur if FormatMessage(...) fails #33
- Using US_DECLARE_SERVICE_INTERFACE with Qt does not work #19
- Fixed documentation of public headers. #165

23.41 v2.1.1 (2014-01-22)

Full Changelog

23.41.1 Fixed

- Resource compiler not found error #11

23.42 v2.1.0 (2014-01-11)

Full Changelog

23.42.1 Changed

- Use the version number from CMakeLists.txt in the manifest file #10

23.42.2 Fixed

- Build fails on Mac OS Mavericks with 10.9 SDK #7
- Comparison of service listener objects is buggy on VS 2008 #9
- Service listener memory leak #8

23.43 v2.0.0 (2013-12-23)

Full Changelog

Major release with backwards incompatible changes. See the migration guide for a detailed list of changes.

23.43.1 Added

- Removed the base class requirement for service objects
- Improved compile time type checking when working with the service registry
- Added a new service factory class for creating multiple service instances based on RFC 195 Service Scopes
- Added ModuleFindHook and ModuleEventHook classes
• Added Service Hooks support
• Added the utility class `us::LDAPProp` for creating LDAP filter strings fluently
• Added support for getting file locations for writing persistent data

23.43.2 Removed

• Removed the output stream operator for `us::Any`

23.43.3 Fixed

• `US_ABI_LOCAL` and symbol visibility for gcc < 4 #6

23.44 v1.0.0 (2013-07-18)

Initial release.

23.44.1 Fixed

• Build fails on Windows with VS 2012 RC due to CreateMutex #5
• usConfig.h not added to framework on Mac #4
• `US_DEBUG` logs even when not in debug mode #3
• Segmentation error after unloading module #2
• Build fails on Ubuntu 12.04 #1
Symbols

- bundle-file, -b
  usResourceCompiler3 command line option, 79
- bundle-name, -n
  usResourceCompiler3 command line option, 79
- compression-level, -c
  usResourceCompiler3 command line option, 79
- help, -h
  usResourceCompiler3 command line option, 79
  usShell3 command line option, 83
- load, -l
  usShell3 command line option, 83
- manifest-add, -m
  usResourceCompiler3 command line option, 79
- out-file, -o
  usResourceCompiler3 command line option, 79
- res-add, -r
  usResourceCompiler3 command line option, 79
- verbose, -V
  usResourceCompiler3 command line option, 79
- zip-add, -z
  usResourceCompiler3 command line option, 79

C

command

  usFunctionAddResources, 85, 86–88
  usFunctionEmbedResources, 85, 86, 87, 88
  usFunctionGenerateBundleInit, 88, 191
  usFunctionGetResourceSource, 86, 87, 87
  cppmicroservices::AbstractWebConsolePlugin (C++ class), 231
  cppmicroservices::AbstractWebConsolePlugin::DoGet
    (C++ function), 232
  cppmicroservices::AbstractWebConsolePlugin::EndResponse
    (C++ function), 233
  cppmicroservices::AbstractWebConsolePlugin::GetCategory
    (C++ function), 232
  cppmicroservices::AbstractWebConsolePlugin::GetCssReferences
    (C++ function), 233
  cppmicroservices::AbstractWebConsolePlugin::GetLabel
    (C++ function), 232
  cppmicroservices::AbstractWebConsolePlugin::GetTitle
    (C++ function), 232
  cppmicroservices::AbstractWebConsolePlugin::GetVariableResolver
    (C++ function), 232
  cppmicroservices::AbstractWebConsolePlugin::IsHtmlRequest
    (C++ function), 232
  cppmicroservices::AbstractWebConsolePlugin::ReadTemplateFile
    (C++ function), 234
  cppmicroservices::AbstractWebConsolePlugin::RenderContent
    (C++ function), 233
  cppmicroservices::AbstractWebConsolePlugin::RenderTopNavigation
    (C++ function), 233
  cppmicroservices::AbstractWebConsolePlugin::SetVariableResolver
    (C++ function), 232
  cppmicroservices::AbstractWebConsolePlugin::StartResponse
    (C++ function), 233
  cppmicroservices::AbstractWebConsolePlugin::TemplateData
    (C++ type), 231
  cppmicroservices::Any (C++ class), 162
  cppmicroservices::Any::Any (C++ function), 162
  cppmicroservices::Any::Empty (C++ function), 164
  cppmicroservices::Any::operator= (C++ function), 163
  cppmicroservices::Any::operator== (C++ function), 163, 164
  cppmicroservices::Any::operator==(C++ function), 163
  cppmicroservices::Any::operator=(C++ function), 162
  cppmicroservices::Any::operator=(C++ function), 164
  cppmicroservices::Any::ToJSON (C++ function), 164
  cppmicroservices::Any::ToString (C++ function), 164
  cppmicroservices::Any::ToStringNoExcept (C++ function), 164
  cppmicroservices::Any::Type (C++ function), 164
  cppmicroservices::any_cast (C++ function), 160, 161
  cppmicroservices::any_cast (C++ function), 167
  cppmicroservices::any_cast (C++ function), 168
  cppmicroservices::any_cast (C++ function), 168
cppmicroservices::AnyMap::AnyMap (C++ function), 165
cppmicroservices::AnyMap::AtCompoundKey (C++ function), 165, 166
cppmicroservices::AnyMap::GetType (C++ function), 165
cppmicroservices::async::AsyncWorkService (C++ class), 219
cppmicroservices::async::AsyncWorkService::~AsyncWorkService (C++ function), 219
cppmicroservices::async::AsyncWorkService::post (C++ function), 219
cppmicroservices::BadAnyCastException (C++ class), 164
cppmicroservices::BadAnyCastException::~BadAnyCastException (C++ function), 165
cppmicroservices::BadAnyCastException::BadAnyCastException (C++ function), 165
cppmicroservices::BadAnyCastException::what (C++ function), 165
cppmicroservices::Bundle (C++ class), 89
cppmicroservices::Bundle::~Bundle (C++ function), 92
cppmicroservices::Bundle::Bundle (C++ function), 92, 102
cppmicroservices::Bundle::c (C++ member), 102
cppmicroservices::Bundle::d (C++ member), 102
cppmicroservices::Bundle::FindResources (C++ function), 97
cppmicroservices::Bundle::GetBundleContext (C++ function), 93
cppmicroservices::Bundle::GetBundleId (C++ function), 93
cppmicroservices::Bundle::GetHeaders (C++ function), 95
cppmicroservices::Bundle::GetLastModified (C++ function), 97
cppmicroservices::Bundle::GetLocation (C++ function), 93
cppmicroservices::Bundle::GetProperties (C++ function), 95
cppmicroservices::Bundle::GetProperty (C++ function), 95
cppmicroservices::Bundle::GetPropertyKeys (C++ function), 96
cppmicroservices::Bundle::GetRegisteredServices (C++ function), 96
cppmicroservices::Bundle::GetResource (C++ function), 96
cppmicroservices::Bundle::GetServicesInUse (C++ function), 96
cppmicroservices::Bundle::GetState (C++ function), 93
cppmicroservices::Bundle::GetSymbol (C++ function), 94
cppmicroservices::Bundle::GetSymbolicName (C++ function), 94
cppmicroservices::Bundle::GetVersion (C++ function), 94
cppmicroservices::Bundle::operator = (C++ function), 92
cppmicroservices::Bundle::operator bool (C++ function), 92
cppmicroservices::Bundle::operator== (C++ function), 92
 cppmicroservices::Bundle::operator== (C++ function), 93
cppmicroservices::Bundle::operator< (C++ function), 92
cppmicroservices::Bundle::Stop (C++ function), 99, 100
cppmicroservices::Bundle::TimeStamp (C++ type), 91
cppmicroservices::Bundle::Uninstall (C++ function), 101
cppmicroservices::BundleActivator (C++ class), 102
cppmicroservices::BundleActivator::~BundleActivator (C++ function), 103
cppmicroservices::BundleActivator::Start (C++ function), 103
cppmicroservices::BundleActivator::Stop (C++ function), 103
 cppmicroservices::BundleContext (C++ class), 103
 cppmicroservices::BundleContext::AddBundleListener (C++ function), 114
cppmicroservices::BundleContext::AddFrameworkListener (C++ function), 114
cppmicroservices::BundleContext::AddServiceListener (C++ function), 112
 cppmicroservices::BundleContext::BundleContext (C++ function), 104
cppmicroservices::BundleContext::GetBundle (C++ function), 105
cppmicroservices::BundleContext::GetBundles (C++ function), 106
 cppmicroservices::BundleContext::GetDataFile (C++ function), 118
 cppmicroservices::BundleContext::GetProperties (C++ function), 105
 cppmicroservices::BundleContext::GetProperty (C++ function), 105
 cppmicroservices::BundleContext::GetPropertyKeys (C++ function), 106
 cppmicroservices::BundleContext::GetRegisteredServices (C++ function), 112
 cppmicroservices::BundleContext::GetServiceReference (C++ function), 110
 cppmicroservices::BundleContext::GetServiceReferences (C++ function), 109
 cppmicroservices::BundleContext::InstallBundles (C++ function), 118
 cppmicroservices::BundleContext::operator = (C++ function), 104
cppmicroservices::BundleContext::operator bool (C++ function), 105
cppmicroservices::BundleContext::operator= (C++ function), 105
cppmicroservices::BundleContext::operator== (C++ function), 104
cppmicroservices::BundleContext::operator< (C++ function), 104
cppmicroservices::BundleContext::RegisterService (C++ function), 106–108
cppmicroservices::BundleContext::RemoveBundleListener (C++ function), 114
cppmicroservices::BundleContext::RemoveFrameworkListener (C++ function), 115
cppmicroservices::BundleContext::RemoveListener (C++ function), 115
cppmicroservices::BundleContext::RemoveServiceListener (C++ function), 115
cppmicroservices::BundleEvent (C++ class), 119
cppmicroservices::BundleEvent::BundleEvent (C++ function), 121
cppmicroservices::BundleEvent::GetBundle (C++ function), 121
cppmicroservices::BundleEvent::GetOrigin (C++ function), 121
cppmicroservices::BundleEvent::GetType (C++ function), 121
cppmicroservices::BundleEvent::operator bool (C++ function), 121
cppmicroservices::BundleEvent::operator== (C++ function), 122
cppmicroservices::BundleEventHook (C++ class), 122
cppmicroservices::BundleEventHook::~BundleEventHook (C++ function), 122
cppmicroservices::BundleEventHook::Event (C++ function), 122
cppmicroservices::BundleFindHook (C++ class), 123
cppmicroservices::BundleFindHook::BundleFindHook (C++ function), 123
cppmicroservices::BundleFindHook::Find (C++ function), 123
cppmicroservices::BundleResource (C++ class), 123
cppmicroservices::BundleResource::~BundleResource (C++ function), 124
cppmicroservices::BundleResource::BundleResource (C++ function), 124
cppmicroservices::BundleResource::GetBaseName (C++ function), 126
cppmicroservices::BundleResource::GetCompleteSuffix (C++ function), 126
cppmicroservices::BundleResource::GetCompressedSize (C++ function), 127
cppmicroservices::BundleResource::GetCrc32 (C++ function), 127
cppmicroservices::BundleResource::GetLastModified (C++ function), 127
cppmicroservices::BundleResource::GetName (C++ function), 125
cppmicroservices::BundleResource::GetResourcePath (C++ function), 126
cppmicroservices::BundleResource::GetSize (C++ function), 127
cppmicroservices::BundleResource::GetSuffix (C++ function), 126
cppmicroservices::BundleResource::IsValid (C++ function), 125
cppmicroservices::BundleResource::operator= (C++ function), 124
cppmicroservices::BundleResource::operator bool (C++ function), 125
cppmicroservices::BundleResource::operator= (C++ function), 125
cppmicroservices::BundleResource::operator== (C++ function), 125
cppmicroservices::BundleResource::operator< (C++ function), 124
cppmicroservices::BundleResourceStream (C++ class), 128
cppmicroservices::BundleResourceStream::BundleResourceStream (C++ function), 128
cppmicroservices::BundleResourceStream::operator= (C++ function), 128
cppmicroservices::BundleResourceStream::operator== (C++ function), 128
cppmicroservices::BundleResource::GetChildren (C++ function), 127
cppmicroservices::BundleResource::GetChildResources (C++ function), 127
cppmicroservices::BundleResource::GetCompleteBaseName (C++ function), 126
cppmicroservices::BundleResource::GetCompleteSuffix (C++ function), 126
cppmicroservices::BundleResource::GetCompressedSize (C++ function), 127
cppmicroservices::BundleResource::GetCrc32 (C++ function), 127
cppmicroservices::BundleResource::GetLastModified (C++ function), 127
cppmicroservices::BundleResource::GetName (C++ function), 125
cppmicroservices::BundleResource::GetResourcePath (C++ function), 126
cppmicroservices::BundleResource::GetSize (C++ function), 127
cppmicroservices::BundleResource::GetSuffix (C++ function), 126
cppmicroservices::BundleVersion (C++ class), 171
cppmicroservices::BundleVersion::BundleVersion (C++ function), 171, 172
cppmicroservices::BundleVersion::Compare (C++ function), 173
cppmicroservices::BundleVersion::EmptyVersion (C++ function), 173
cppmicroservices::BundleVersion::GetMajor (C++ function), 172
cppmicroservices::BundleVersion::GetMicro (C++ function), 172
cppmicroservices::BundleVersion::GetMinor (C++ function), 172
cppmicroservices::BundleVersion::GetQualifier (C++ function), 172

cppmicroservices::framework::dto::ServiceReferenceDTO::bundle (C++ member), 206
cppmicroservices::framework::dto::ServiceReferenceDTO::id (C++ member), 206
cppmicroservices::framework::dto::ServiceReferenceDTO::properties (C++ member), 206
cppmicroservices::framework::dto::ServiceReferenceDTO::usingBundles (C++ member), 206
cppmicroservices::Framework::Framework (C++ function), 129
cppmicroservices::Framework::Init (C++ function), 129
cppmicroservices::Framework::operator= (C++ function), 129
cppmicroservices::Framework::WaitForStop (C++ function), 129
cppmicroservices::FrameworkEvent (C++ class), 130
cppmicroservices::FrameworkEvent::FrameworkEvent (C++ function), 131
cppmicroservices::FrameworkEvent::GetBundle (C++ function), 132
cppmicroservices::FrameworkEvent::GetMessage (C++ function), 132
cppmicroservices::FrameworkEvent::GetThrowable (C++ function), 132
cppmicroservices::FrameworkEvent::GetType (C++ function), 132
cppmicroservices::FrameworkEvent::operator bool (C++ function), 131
cppmicroservices::FrameworkFactory (C++ class), 178
cppmicroservices::FrameworkFactory::NewFramework (C++ function), 179
cppmicroservices::GetBundleContext (C++ function), 179
cppmicroservices::HttpServletRequest (C++ class), 223
cppmicroservices::HttpServletRequest::~HttpServletRequest (C++ function), 223
cppmicroservices::HttpServletRequest::GetAcceptHeader (C++ function), 224
cppmicroservices::HttpServletRequest::GetAttribute (C++ function), 223
cppmicroservices::HttpServletRequest::GetAttributeNames (C++ function), 223
cppmicroservices::HttpServletRequest::GetContentLength (C++ function), 223
cppmicroservices::HttpServletRequest::GetContentType (C++ function), 223
cppmicroservices::HttpServletRequest::GetContextPath (C++ function), 223
cppmicroservices::HttpServletRequest::GetDateHeader (C++ function), 223
cppmicroservices::HttpServletRequest::GetHeader (C++ function), 223
cppmicroservices::HttpServletRequest::GetHeaderNames (C++ function), 223
cppmicroservices::HttpServletRequest::GetHeaders (C++ function), 223
cppmicroservices::HttpServletRequest::GetLocalName (C++ function), 223
cppmicroservices::HttpServletRequest::GetLocalPort (C++ function), 223
cppmicroservices::HttpServletRequest::GetMethod (C++ function), 223
cppmicroservices::HttpServletRequest::GetPathInfo (C++ function), 223
cppmicroservices::HttpServletRequest::GetProtocol (C++ function), 223
cppmicroservices::HttpServletRequest::GetQueryString (C++ function), 223
cppmicroservices::HttpServletRequest::GetRemoteHost (C++ function), 223
cppmicroservices::HttpServletRequest::GetRemotePort (C++ function), 223
cppmicroservices::HttpServletRequest::GetRequestUri (C++ function), 223
cppmicroservices::HttpServletRequest::GetRequestUrl (C++ function), 223
cppmicroservices::HttpServletRequest::GetScheme (C++ function), 223

cppmicroservices::HttpServletRequest::GetServerName (C++ function), 223
cppmicroservices::HttpServletRequest::GetServerPort (C++ function), 223
cppmicroservices::HttpServletRequest::GetServletContext (C++ function), 223
cppmicroservices::HttpServletRequest::GetServletPath (C++ function), 223
cppmicroservices::HttpServletRequest::HttpServletRequest (C++ function), 223
cppmicroservices::HttpServletRequest::operator= (C++ function), 223
cppmicroservices::HttpServletRequest::RemoveAttribute (C++ function), 224
cppmicroservices::HttpServletRequest::SetAttribute (C++ function), 224
cppmicroservices::HttpServletResponse (C++ class), 224
cppmicroservices::HttpServletResponse::AddHeader (C++ function), 224
cppmicroservices::HttpServletResponse::AddIntHeader (C++ function), 224
cppmicroservices::HttpServletResponse::ContainsHeader (C++ function), 224
cppmicroservices::HttpServletResponse::d (C++ member), 227
cppmicroservices::HttpServletResponse::FlushBuffer (C++ function), 224
cppmicroservices::HttpServletResponse::GetBufferSize (C++ function), 224
cppmicroservices::HttpServletResponse::GetCharacterEncoding (C++ function), 224
cppmicroservices::HttpServletResponse::GetContentType (C++ function), 224
cppmicroservices::HttpServletResponse::GetHeader (C++ function), 224
cppmicroservices::HttpServletResponse::GetOutputStream (C++ function), 224
cppmicroservices::HttpServletResponse::GetOutputStreamBuffer (C++ function), 227
cppmicroservices::HttpServletResponse::GetStatus (C++ function), 224
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_ACCEPTED (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_BAD_GATEWAY (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_BAD_REQUEST (C++ member), 226
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_CONTINUE (C++ member), 226
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_CREATED (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_EXPECTATION_FAILED (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_FORBIDDEN (C++ member), 226
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_FOUND (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_GATEWAY_TIMEOUT (C++ member), 227
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_GONE (C++ member), 226
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_HTTP_VERSION_NOT_SUPPORTED (C++ member), 227
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_INTERNAL_SERVER_ERROR (C++ member), 227
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_LENGTH_REQUIRED (C++ member), 226
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_METHOD_NOT_ALLOWED (C++ member), 226
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_MOVED_PERMANENTLY (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_MOVED_TEMPORARILY (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_MULTIPLE_CHOICES (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_NO_CONTENT (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_NON_AUTHORITATIVE_INFORMATION (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_OK (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_PARTIAL_CONTENT (C++ member), 225
cppmicroservices::HttpServletResponse::HttpServletRequest::SC_PAYMENT_REQUIRED (C++ member), 226
Index 259
cppmicroservices::HttpServletResponse::SC_PRECONDITION_FAILED (C++ member), 226
cppmicroservices::HttpServletResponse::SC_PROXY_AUTHENTICATION_REQUIRED (C++ member), 226
cppmicroservices::HttpServletResponse::SC_REQUEST_ENTITY_TOO_LARGE (C++ member), 226
cppmicroservices::HttpServletResponse::SC_REQUEST_TIMEOUT (C++ member), 226
cppmicroservices::HttpServletResponse::SC_REQUEST_URI_TOO_LONG (C++ member), 226
cppmicroservices::HttpServletResponse::SC_REQUESTED_RANGE_NOT_SATISFIABLE (C++ member), 227
cppmicroservices::HttpServletResponse::SC_RESET_CONTENT (C++ member), 225
cppmicroservices::HttpServletResponse::SC_SEE_OTHER (C++ member), 225
cppmicroservices::HttpServletResponse::SC_SERVICE_UNAVAILABLE (C++ member), 227
cppmicroservices::HttpServletResponse::SC_SWITCHING_PROTOCOLS (C++ member), 225
cppmicroservices::HttpServletResponse::SC_TEMPORARY_REDIRECT (C++ member), 226
cppmicroservices::HttpServletResponse::SC_UNAUTHORIZED (C++ member), 226
cppmicroservices::HttpServletResponse::SC_UNSUPPORTED_MEDIA_TYPE (C++ member), 226
cppmicroservices::HttpServletResponse::SC_USE_PROXY (C++ member), 225
cppmicroservices::HttpServletResponse::SendError (C++ function), 225
cppmicroservices::HttpServletResponse::SendRedirect (C++ function), 225
cppmicroservices::HttpServletResponse::SetBufferSize (C++ function), 224
cppmicroservices::HttpServletResponse::SetCharacterEncoding (C++ function), 224
cppmicroservices::HttpServletResponse::SetContentLength (C++ function), 224
cppmicroservices::HttpServletResponse::SetContentType (C++ function), 224
cppmicroservices::HttpServletResponse::SetDateHeader (C++ function), 224
cppmicroservices::HttpServletResponse::SetHeader (C++ function), 224
cppmicroservices::HttpServletResponse::SetIntHeader (C++ function), 224
cppmicroservices::HttpServletResponse::SetOutputStreamBuffer (C++ function), 227
cppmicroservices::HttpServletResponse::SetStatus (C++ function), 224
cppmicroservices::LDAPFilter::LDAPFilter (C++ function), 181
cppmicroservices::LDAPFilter::Match (C++ function), 181, 182
cppmicroservices::LDAPFilter::MatchCase (C++ function), 182
cppmicroservices::LDAPFilter::operator bool (C++ function), 181
cppmicroservices::LDAPFilter::operator= (C++ function), 182
cppmicroservices::LDAPFilter::operator== (C++ function), 182
cppmicroservices::LDAPFilter::ToString (C++ function), 182
cppmicroservices::LDAPFilter::d (C++ member), 183
cppmicroservices::LDAPFilter::LDAPFilter (C++ function), 181
cppmicroservices::LDAPFilter::Match (C++ function), 181, 182
cppmicroservices::LDAPFilter::MatchCase (C++ function), 182
cppmicroservices::LDAPFilter::operator bool (C++ function), 181
cppmicroservices::LDAPFilter::operator= (C++ function), 182
cppmicroservices::LDAPFilter::operator== (C++ function), 182
cppmicroservices::LDAPFilter::operator>= (C++ function), 180, 183, 184
cppmicroservices::LDAPFilter::operator<= (C++ function), 180, 184
cppmicroservices::LDAPProp (C++ class), 183
cppmicroservices::LDAPProp::Approx (C++ function), 180, 184
cppmicroservices::LDAPProp::LDAPProp (C++ function), 183
cppmicroservices::LDAPProp::operator= (C++ function), 179, 180, 183
cppmicroservices::LDAPProp::operator LDAPPropExpr (C++ function), 183
cppmicroservices::LDAPProp::operator== (C++ function), 179, 183
cppmicroservices::LDAPProp::operator>= (C++ function), 180, 183, 184
cppmicroservices::LDAPProp::operator<= (C++ function), 180, 184
cppmicroservices::LDAPProp::toString (C++ function), 182
cppmicroservices::LDAPProp::toString (C++ function), 182
cppmicroservices::logservice::LogService (C++ class), 217
cppmicroservices::logservice::LogService::~LogService (C++ function), 218
cppmicroservices::MakeInterfaceMap (C++ class), 140
cppmicroservices::MakeInterfaceMap::MakeInterfaceMap (C++ function), 141
cppmicroservices::MakeInterfaceMap::operator InterfaceMapConstPtr (C++ function), 141
cppmicroservices::MakeInterfaceMap::operator InterfaceMapPtr (C++ function), 141
cppmicroservices::operator== (C++ function), 130
cppmicroservices::operator<< (C++ function), 89, 119, 123, 130, 134, 146, 171, 179
cppmicroservices::PrototypeServiceFactory (C++ class), 132
cppmicroservices::PrototypeServiceFactory::GetService (C++ function), 133
cppmicroservices::PrototypeServiceFactory::UngetService (C++ function), 133
cppmicroservices::ref_any_cast (C++ function), 161
cppmicroservices::service::cm::Configuration (C++ function), 161

cppmicroservices::service::component::runtime::dto::ReferenceDTO::unbind
(C++ member), 196
cppmicroservices::service::component::runtime::dto::ReferenceDTO::target
(C++ member), 196
cppmicroservices::service::component::runtime::dto::ReferenceDTO::scope
(C++ member), 196
cppmicroservices::service::component::runtime::dto::ReferenceDTO::policyOption
(C++ member), 196
cppmicroservices::service::component::runtime::dto::ReferenceDTO::name
(C++ member), 196
cppmicroservices::service::component::runtime::dto::ReferenceDTO::interfaceName
(C++ member), 196
cppmicroservices::service::component::runtime::dto::ReferenceDTO::cardinality
(C++ member), 196
cppmicroservices::service::component::runtime::dto::ReferenceDTO::bind
(C++ member), 200
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::serviceInterfaces
(C++ class), 200
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::scope
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::references
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::properties
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::name
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::modified
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::implementationClass
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::defaultEnabled
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::bundle
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::activate
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentDescriptionDTO::isComponentEnabled
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentConfigurationDTO::unsatisfiedReferences
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentConfigurationDTO::state
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentConfigurationDTO::satisfiedReferences
(C++ member), 201
cppmicroservices::service::component::runtime::dto::ComponentConfigurationDTO::properties
(C++ member), 201
...
cppmicroservices::ServiceException (C++ class), 136
cppmicroservices::ServiceException::~ServiceException (C++ function), 137
cppmicroservices::ServiceException::GetType (C++ function), 137
cppmicroservices::ServiceException::operator= (C++ function), 137
cppmicroservices::ServiceFactory (C++ class), 137
cppmicroservices::ServiceFactory::GetService (C++ function), 138
cppmicroservices::ServiceFactory::UngetService (C++ function), 138
cppmicroservices::ServiceFactory::ServiceFactory (C++ function), 138
cppmicroservices::ServiceFindHook (C++ class), 139
cppmicroservices::ServiceFindHook::~ServiceFindHook (C++ function), 139
cppmicroservices::ServiceFindHook::Find (C++ function), 139
cppmicroservices::ServiceListenerHook (C++ class), 142
cppmicroservices::ServiceListenerHook::~ServiceListenerHook (C++ function), 142
cppmicroservices::ServiceListenerHook::ListenerInfo (C++ function), 142
cppmicroservices::ServiceListenerHook::ListenerInfo::GetBundleContext (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::GetFilter (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::IsNull (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::IsRemoved (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::ListenerInfo (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::operator= (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::operator== (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::ListenerInfo (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::GetBundleContext (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::GetFilter (C++ function), 143
cppmicroservices::ServiceListenerHook::ListenerInfo::IsNull (C++ function), 143
cppmicroservices::ServiceListenerHook::Removed (C++ function), 143
cppmicroservices::ServiceObjects (C++ class), 144
cppmicroservices::ServiceObjects::GetService (C++ function), 144, 145
cppmicroservices::ServiceObjects::GetServiceReference (C++ function), 145
cppmicroservices::ServiceReference (C++ class), 146
cppmicroservices::ServiceReference::operator= (C++ function), 146
cppmicroservices::ServiceReference::ServiceReference (C++ function), 146
cppmicroservices::ServiceReference::ServiceReference (C++ function), 146
cppmicroservices::ServiceReference::ServiceReference (C++ function), 146
cppmicroservices::ServiceReferenceBase (C++ class), 146
cppmicroservices::ServiceReferenceBase::GetBundle (C++ function), 147
cppmicroservices::ServiceReferenceBase::GetInterfaceId (C++ function), 147
cppmicroservices::ServiceReferenceBase::GetProperty (C++ function), 147
cppmicroservices::ServiceReferenceBase::GetPropertyKeys (C++ function), 147
cppmicroservices::ServiceReferenceBase::GetUsingBundles (C++ function), 148
cppmicroservices::ServiceReferenceBase::IsConvertibleTo (C++ function), 148
cppmicroservices::ServiceReferenceBase::operator bool (C++ function), 148
cppmicroservices::ServiceReferenceBase::ServiceRegistration (C++ class), 149
cppmicroservices::ServiceRegistration::GetReference (C++ function), 149
cppmicroservices::ServiceRegistration::ServiceRegistration (C++ function), 149
cppmicroservices::ServiceRegistrationBase (C++ class), 150
cppmicroservices::ServiceRegistrationBase::~ServiceRegistrationBase (C++ function), 150
cppmicroservices::ServiceRegistrationBase::GetReference (C++ function), 151
cppmicroservices::ServiceRegistrationBase::ServiceRegistration (C++ function), 150
cppmicroservices::ServiceRegistrationBase::ServiceRegistration (C++ function), 150
cppmicroservices::ServiceRegistrationBase::ServiceRegistration (C++ function), 150
cppmicroservices::ServiceRegistrationBase (C++ class), 150
cppmicroservices::ServiceRegistrationBase::ServiceRegistration (C++ function), 151
cppmicroservices::ServiceRegistrationBase::ServiceRegistration (C++ function), 151
cppmicroservices::ServiceRegistrationBase::ServiceRegistration (C++ function), 151
cppmicroservices::ServiceRegistrationBase::operator bool (C++ function), 150
cppmicroservices::ServiceRegistrationBase::operator bool (C++ function), 150
cppmicroservices::ServiceRegistrationBase::operator= (C++ function), 152
cppmicroservices::ServiceRegistrationBase::operator== (C++ function), 152
<table>
<thead>
<tr>
<th>cppmicroservices::ServletContext::GetMimeType (C++ function),</th>
<th>229</th>
</tr>
</thead>
<tbody>
<tr>
<td>cppmicroservices::SharedLibrary (C++ class),</td>
<td>186</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::~SharedLibrary (C++ function),</td>
<td>186</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::GetFilePath (C++ function),</td>
<td>187</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::GetHandle (C++ function),</td>
<td>188</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::GetLibraryPath (C++ function),</td>
<td>187</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::GetName (C++ function),</td>
<td>187</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::GetPrefix (C++ function),</td>
<td>188</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::GetSuffix (C++ function),</td>
<td>188</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::IsLoaded (C++ function),</td>
<td>188</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::Load (C++ function),</td>
<td>186</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::operator= (C++ function),</td>
<td>186</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::SetFilePath (C++ function),</td>
<td>187</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::SetLibraryPath (C++ function),</td>
<td>187</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::SetName (C++ function),</td>
<td>187</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::SetPrefix (C++ function),</td>
<td>188</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::SetSuffix (C++ function),</td>
<td>188</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::SharedLibrary (C++ function),</td>
<td>186</td>
</tr>
<tr>
<td>cppmicroservices::SharedLibrary::Unload (C++ function),</td>
<td>186</td>
</tr>
<tr>
<td>cppmicroservices::ShellService (C++ class),</td>
<td>239</td>
</tr>
<tr>
<td>cppmicroservices::ShellService::~ShellService (C++ function),</td>
<td>239</td>
</tr>
<tr>
<td>cppmicroservices::ShellService::ExecuteCommand (C++ function),</td>
<td>239</td>
</tr>
<tr>
<td>cppmicroservices::ShellService::GetCompletions (C++ function),</td>
<td>239</td>
</tr>
<tr>
<td>cppmicroservices::ShellService::ShellService (C++ function),</td>
<td>239</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap (C++ class),</td>
<td>188</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::empty (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::end (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::equal_range (C++ function),</td>
<td>190</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::erase (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::find (C++ function),</td>
<td>190</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::lower_bound (C++ function),</td>
<td>190</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::max_size (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::operator[] (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::ShrinkableMap (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::size (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap::operator= (C++ function),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::const_iterator (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::const_reference (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::container_type (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::iterator (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::key_type (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::mapped_type (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::reference (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::size_type (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableMap&lt;Key, T&gt;::value_type (C++ type),</td>
<td>189</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableVector (C++ class),</td>
<td>190</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableVector::at (C++ function),</td>
<td>190</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableVector::back (C++ function),</td>
<td>191</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableVector::begin (C++ function),</td>
<td>190</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableVector::clear (C++ function),</td>
<td>191</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableVector::empty (C++ function),</td>
<td>191</td>
</tr>
<tr>
<td>cppmicroservices::ShrinkableVector::end (C++ function),</td>
<td>191</td>
</tr>
</tbody>
</table>

**Index**

265