Balau core C++ library

User manual

Version 2019.7.1

Bora Software

BORA®

CONTENTS

Balau core C++ library

Overview

Balau is a C++ application framework designed for enterprise quality C++ software development.

Following the recent revisions of the language, C++ has matured to become an attractive candidate for rapid enterprise quality application development. Balau provides tools designed to support the rapid development of high performance C++ enterprise applications.

Balau builds on the foundations of the Boost and ICU projects, and focuses on using modern C++17 features and the standard unicode string classes.

The library has been conceived for the development of applications that have a dependency injection based architecture, have complex logging requirements, and will be developed with a test driven development methodology. Balau has also been designed to provide an application framework for Unicode aware C++ software applications.

Four key components of the Balau library are the injector, the environment configuration framework, the logging system, and the test runner. In this respect, part of Balau is a C++ equivalent to the de facto standard Java based application development components consisting of *Guice*/*Spring* for dependency injection and environment configuration, *Log4j*/*Slf4j*/*Logback* for logging, and *JUnit*/*TestNG* for testing.

In addition to the injector, environment configuration, logger, and test runner, Balau provides a set of components and utilities with simple APIs, including an HTTP/WebSocket web application framework.

Links

User manual (this document): https://borasoftware.com/doc/balau/latest/manual.

API documentation: https://borasoftware.com/doc/balau/latest/api.

Main Git repository: https://github.com/borasoftware/balau.

Intended audience

The following questions may be useful in order to determine if the Balau library is suitable for your requirements.

- Do you wish to develop a software application in C++ using modern best practices?
- Will the code be based on C++17 or a later specification?

- Will you be using the Boost libraries?
- Do you wish to structure your application via a dependency injection based architecture?
- Do you need to configure the application injector differently for multiple environments?
- Do you require a flexible logging system?
- Does the development team wish to use a type safe, class based, in process / out of process test framework?
- Does the development team prefer user friendly canned utilities and utility APIs?
- Do you require HTTP/WebSocket connectivity and an integrated web application framework?
- Does the development team wish to develop a recursive descent language parser in pure C++?

Themes

The main themes currently covered by the Balau library are as follows. Documentation for each component / utility may be accessed from the drop down menu at the top of this page.

- dependency injection with application and environment configuration
- Hierarchical, typed environment configuration
- Logging system, configured via environment configuration
- Test framework
- Unified to-string functions
- Unified from-string functions
- Unicode character and string utilities
- Unified resource identification and resource access
- Data structures
- Shared memory data structures
- Concurrent data structures
- Hand written language parser utilities
- Hierarchical property file format and corresponding hand written parser
- HTTP/WebSocket clients and server
- HTTP/WebSocket web application framework

- Utilities (system, compression, files, hashing, streams, strings, etc.)

Documentation pages

Developer manual

Each documentation page provided in the top menu contains documentation on the component or utility, with the following structure:

- overview;
- quick start guide;
- optional detailed documentation sections for complex components;
- an optional design discussion.

Application developers should be able to get up to speed on each component / utility by reading the overview and quick start guide. They can subsequently refer to the detailed documentation later on when more advanced use of the component / utility is required.

The Balau documentation is written in BDML, and can be loaded directly in a web browser with the included BDML XSLT stylesheet*, or translated to HTML via the *BalauManual* make target (xsltproc is required). An HTML translation of the documentation can also be viewed online at https://borasoftware.com/doc/balau/latest/manual.

API documentation

API documentation can be generated by running the *BalauApiDoc* make target (doxygen is required). The API documentation can also be viewed online at https://borasoftware.com/doc/balau/latest/api.

Dependencies

In addition to the C++17 standard library, Balau relies on two main third party libraries and three utility libraries.

The first main dependency is ICU, which provides Unicode support functions. The second main dependency is the Boost library. Boost is used for low level essential utilities and complex, low level components not found in the C++ standard library.

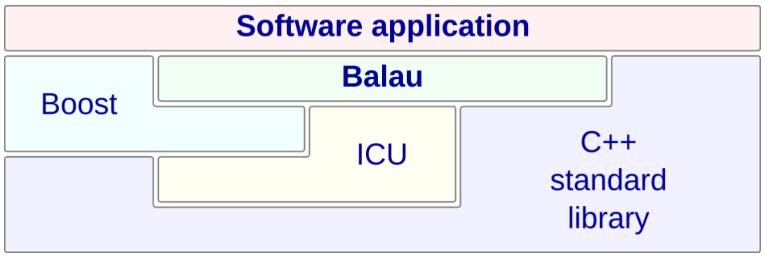
The three utility library dependencies are *zlib*, *libzip* and *curl*. The first two utility libraries provide low level compression support for the compression utilities in Balau. The third utility library provides network protocol support in the network classes.

The only other dependencies used are standard dependencies on each supported platform.

Balau includes a small number of third party utility libraries in its source code release. As these are contained within the Balau source code tree, they are not external dependencies and thus do not require linkage. These third party libraries are contained within the *ThirdParty* folder. As these libraries are supplied with Balau, application developers can also use them directly if required. The libraries have been namespaced within the *Balau* outer namespace in order to avoid potential clashes.

Application structure

The high level structure of a software application or library based on Balau is shown in the diagram below.



License

Balau is licensed under the Boost Software License Version 1.0.

Supported platforms

C++ version

Balau requires a compiler that is compliant with C++17 or a later version of the specification, with certain exceptions. Notably, guaranteed copy elision is not required (since version 2019.5.1) and *boost::string_view* is used as a substitute for *std::string_view* if *std::string_view* is not available.

These exceptions allow Balau to be built with the partially C++17 compliant GCC version 6.

Operating systems

Balau has been developed and tested on 64 bit Linux with the GCC and Clang compilers. A port to Windows 7/10 is planned.

Other Posix compliant platforms that are supported by Boost and ICU may work if they comply with the primitive type size assertion checks in the *StdTypes.hpp* header:

```
static_assert(CHAR_BIT           == 8);
static_assert(sizeof(short)      == 2);
static_assert(sizeof(int)        == 4);
static_assert(sizeof(long long) == 8);
```

Due to the size difference between long integers in different common data models, the *long* and *unsigned long* integer types are not used in the library other than when a dependency requires a value of one of those types. Instead, use of the *long long* and *unsigned long long* integer types allows the commonly accepted data models to be supported without primitive type size conflicts.

CPU architectures

Balau has been developed and tested on x86-64.

Concurrent code in the library uses the C++ 11 atomic operations library. Consequently, the library should be free of data races on all platforms that have a standards compliant C++17 compiler and standard library.

Building

Balau uses the CMake build system. See the building page for information on building Balau and its dependencies.

The source code includes a set of unit tests, implemented with the Balau test runner. After building the library, the tests may be run by launching the *BalauTests* application. The tests can also be used as an aid in getting up to speed quickly with each feature in the library.

Contributing

The core principal of the Balau core C++ library is to provide a user friendly C++ application framework on top of Boost and ICU, on which complex Unicode based C++ software applications may be created. To achieve this aim, the library contains a set of core application components (injector, environment configuration framework, logger, test runner) that form the basis of a complex C++ software application, and a set of utilities with straightforward APIs.

The library is in active development. Many of the themes can be expanded to cover a greater breadth of features and utilities.

Pull requests with additional components and utilities are welcome. Some guidelines are included on the contributing page. A current list of planned development is available on the planned features page.

** Configuration of the browser's security settings may be required. See the documentation on direct loading fo*r
more information.

APPLICATION

Injector

Overview

Introduction

A C++ dependency injection framework. The injector is configured via templated binding functions and provides get-instance methods for non-polymorphic and polymorphic values, references, thread-local and non-thread-local singletons. The code based application configuration mechanism is performed via implementations of the *ApplicationConfiguration* base class. Binding declaration calls within application configuration classes define non-polymorphic value (termed *value*), polymorphic value (termed *unique*), reference, and singleton bindings.

The dependency injection framework also integrates with the hierarchical property framework . This allows typed and untyped (*std::string*) environment properties (simple and composite) to be created in the injector via implementations of the *EnvironmentConfiguration* class.

Using this approach, multiple application processes (one set of processes per application environment) may be run from the same injector configuration. Each environment is configured according to the environment's property file(s) and the environment configuration is validated and loaded into the injector by the meta-configuration contained in the implementations of the *EnvironmentConfiguration* class.

Dependency injection

Using a dependency injection approach for the structural wiring of a software application is useful when the complexity of the application reaches a certain threshold. As an application's source code becomes larger and more complex, manual management of dependencies becomes overly complicated and error prone.

In this respect, using one or more injectors in the design of a software application reduces complexity, concentrates structural wiring into a concise set of declarations, and delegates object lifetime management of long lived objects to the injection framework. The dependency injection paradigm also facilitates isolating a class for unit testing, by allowing mocked or stubbed dependencies to be supplied to an instance of the class being tested.

Balau injector

The Balau injector provides a constructor injection paradigm, where the constructors of injectable classes are the populating mechanism of injected instances. Configuration of the injector dependency graph is performed within one or more configuration classes that are specified during injector instantiation. Configuration of each injectable class is achieved via an injector macro that specifies the dependencies that the injector will provide during instantiation of the class.

Templated binding calls are used to provide type information at configuration time. Binding calls are available for the specification of values, prototypes, instances, references, providers, thread-local singletons, singletons, and provided singletons.

In addition to providing configured dependencies, the injector can provide itself as a dependency when the shared *Injector* class is requested. This allows complex injectable classes to use the injector directly, in addition to their standard dependencies. Differently configured injectors can also be injected into a service at runtime, by using injector hierarchies.

There are four types of instance provided by the injector:

- non-polymorphic **value** instances, provided directly as type *ValueT*;
- polymorphic **unique** value instances, provided inside *std::unique_ptr<BaseT>* or *std::unique_ptr<BaseT, DeleterT>* containers;
- polymorphic non-const and const **reference** instances, provided directly as types *BaseT &* and *const BaseT &*;
- polymorphic non-const and const **shared** instances, provided inside *std::shared_ptr<BaseT>* and *std::shared_ptr<const BaseT>* containers.

Each meta-type has its own instantiation semantics:

- **value** instances are stack based non-polymorphic instances created via copy construction or copy elision;
- **unique** instances are new heap based polymorphic instances;
- **reference** instances are const or non-const references provided to the injector via the application configuration;
- **shared** instances are singletons or thread local singletons, either instantiated by the injector or provided to the injector via the application configuration.

The Balau injector is designed to allow arbitrary injector hierarchies to be arranged at runtime and obtained via an instance of the simple (non-template) *Injector* class. This includes injection of the injector into complex injectable classes that require direct access to it. Bindings may also be named, allowing identically typed but differently named bindings to be created and looked up dynamically.

In order to ensure that binding issues are caught before injectors are used, each constructed injector runs a dependency validation phase during instantiation. During this validation phase, the injector constructs a dependency graph of all registered interfaces, classes, providers, and relationships. The required bindings of each dependency are verified. Dependency cycle analysis is also performed. This validation ensures that binding issues are exposed during injector instantiation, allowing simple unit tests to be constructed to test the structural wiring of each the application injector configuration.

Quick start

#include <Balau/Application/Injector.hpp>

#include <Balau/Application/Injectable.hpp>

The steps involved in creating a software application based on the Balau injector are:

1. create one or more injector configuration classes;
2. annotate injector aware classes with injection macros that define the constructor injection semantics.

When creating an injector, the injector factory function takes one or more injector configuration classes. There are two types of injector configuration:

- application configuration;
- environment configuration.

Application configuration defines the fixed application binding definitions for values, instance creation, references, thread-local singletons, and singletons. Environment configuration defines requirements and type information for environment specific value bindings (simple and composite), created from environment specific properties files.

Both types of configuration are defined by creating a class containing an implementation of the *configure* method. The code contained within the *configure* method is different for the two types of configuration.

Application configuration

Application configuration is defined by inheriting from the *ApplicationConfiguration* class and implementing the *configure* method with binding calls.

Each binding is specified by a two part fluent call chain, defined within the *configure* method. The first call in a binding call chain provides the binding's instance type and optional UTF-8 string name. The second call in the chain defines the binding meta-type and any additional type, object, or provider information required.

```
// An example injector configuration class.
class Configuration : public ApplicationConfiguration {
    public: void configure() const override {
        bind<Base>().toSingleton<Derived>();
        bind<Base2>().toUnique<Derived2>();
        bind<Base2>("alternative").toUnique<Derived3>();
    }
};
```

There following binding calls are available.

Binding type	Description
toValue()	Bind a concrete class.
toValue(ValueT)	Bind a prototype value.
toValueProvider(std::function)	Bind a value provider function.
toValueProvider<ProviderT>()	Bind an injectable value provider class.
toValueProvider<ProviderT>(std::shared_ptr<ProviderT>)	Bind an injectable value provider instance.
toUnique<DerivedT>()	Bind an interface to a concrete class.
toUniqueProvider(std::function)	Bind an interface to a polymorphic provider function.
toUniqueProvider<ProviderT>()	Bind an interface to an injectable unique pointer provider class.
toUniqueProvider<ProviderT>(std::shared_ptr<ProviderT>)	Bind an interface to a provided unique pointer provider instance.
toReference(BaseT &)	Bind a reference type to the supplied reference object.
toThreadLocal<DerivedT>()	Bind an interface to a thread-local, lazy concrete singleton class.
toThreadLocal()	Bind a thread-local, lazy concrete singleton class.
toSingleton<DerivedT>()	Bind an interface to a lazy concrete singleton class.
toSingleton()	Bind a lazy concrete singleton class.
toSingleton(std::shared_ptr<BaseT>)	Bind an interface to the supplied singleton object.
toSingleton(BaseT *)	Bind an interface to the supplied singleton object via pointer container initialisation style syntax.
toSingletonProvider<ProviderT>()	Bind an interface to an injectable singleton provider class.
toSingletonProvider<ProviderT>(std::shared_ptr<ProviderT>)	Bind an interface to a provided singleton provider instance.
toEagerSingleton<DerivedT>()	Bind an interface to a concrete eager singleton class.
toEagerSingleton()	Bind a concrete eager singleton class.

Each of the binding types belongs to one or two of six meta-types. The meta-types are:

- ***value*** - non-polymorphic, stack based, instantiated instances;
- ***unique*** - polymorphic, heap based, instantiated instances;
- ***reference*** - polymorphic, provided reference objects;
- ***const reference*** - polymorphic, provided const reference objects;
- ***shared*** - polymorphic, heap based shared objects (thread-local and singleton).

- ***const shared*** - polymorphic, heap based shared const objects (thread-local and singleton).

Bindings that provide references or shared instances can be non-const or const. The meta-type is determined via the presence or absence of a const qualifier in the specified type in the binding call.

The polymorphic, heap based, instantiated instance meta-type is termed *unique* instead of *instance* in order to avoid confusion with the universal *getInstance* methods that access instances from all meta-types according to the full specified type. These calls are discussed later in the Injector usage section.

Environment configuration

Environment configuration is defined by one or two methods:

- inheriting from the *EnvironmentConfiguration* class and implementing the *configure* method with environment property type and requirements declaration calls;
- instantiating the *EnvironmentConfiguration* class directly and by supplying one or more type specification source files.

For more information on defining environment configurations via environment properties within an injector based application, refer to the Environment chapter.

Injection macros

In order that classes take part in dependency injection, an injector macro needs to be added to each of their declarations. There are three types of injector macro available.

Macro	Description
BalauInjectConstruct BalauInjectConstructNamed	Specify the class' direct dependency fields and implicitly create an injectable constructor.
BalauInject BalauInjectNamed	Specify the class' direct or indirect dependency fields. Do not implicitly create an injectable constructor.
BalauInjectTypes BalauInjectNamedTypes	Specify the types of the class' dependencies to be injected. Do not implicitly create an injectable constructor.

All macros take an initial parameter which is the class name. The named versions of the macros take the names of the dependencies in addition to the field names or types.

The choice of which macro to use depends on whether the injectable class' dependencies correspond to direct / indirect fields, or whether one or more of the dependencies will be used in some temporary way instead of being assigned to a field. If the former is the case, then the *BalauInjectConstruct* / *BalauInjectConstructNamed* or *BalauInject* / *BalauInjectNamed* macros can be used. If the latter is the case, then the *BalauInjectTypes* / *BalauInjectNamedTypes* macro should be used, as it will not be possible to automatically determine the types of the dependencies.

The choice of whether to use the *BalauInjectConstruct* / *BalauInjectConstructNamed* or the *BalauInject* / *BalauInjectNamed* macros when all dependencies are being assigned to fields depends on whether all the dependencies are direct or not, and whether all assignments in the constructor initialisation list are simple or not. If they are direct and simple, then one of the *BalauInjectConstruct* / *BalauInjectConstructNamed* macros can be used. Otherwise, the injectable constructor should be manually written and one of the *BalauInject* / *BalauInjectNamed* macros used.

The chosen injector macro should be placed within the injectable class' declaration.

```
//
// ////// BalauInjectConstruct macro //////
//
// Specify the injected dependencies via the class fields
// and implicitly create an injectable constructor.
//
class Derived2 : public Base2 {
    private: std::shared_ptr<Base> dependency;

    BalauInjectConstruct(Derived2, dependency)

    public: void foo() override;
};

//
// ////////// BalauInject macro ///////////
//
// Specify the injected dependencies via the class fields.
//
class Derived2 : public Base2 {
    private: std::shared_ptr<Base> dependency;

    BalauInject(Derived2, dependency)

    // Explicitly created injectable constructor.
    public: Derived2(std::shared_ptr<Base> dependency_)
        : dependency(std::move(dependency_)) {}

    public: void foo() override;
};

//
// //////// BalauInjectTypes macro ////////
//
// Specify the injected dependencies' types.
//
class Derived2 : public Base {
    private: std::shared_ptr<Base> dependency;

    BalauInjectTypes(Derived2, std::shared_ptr<Base>)

    // Explicitly created injectable constructor.
    public: Derived2(std::shared_ptr<Base> dependency_)
        : dependency(std::move(dependency_)) {}

    public: void foo() override;
};
```

The alternative *Named* macros take dependency names. When these macros are used, all dependency names must be specified. Empty names must be specified with empty string literals "".

```
//
// Example with a dependency name.
//
class Derived2 : public Base {
    private: std::shared_ptr<Base> dependency;

    BalauInjectConstructNamed(Derived2, dependency, "myDependency")

    public: void foo() override;
};
```

Injector usage

The injector is created by calling the *Injector::create(Conf(), ...)* function.

```
auto injector = Injector::create(Configuration(), ExtraConfiguration());
```

An alternative create function is also available, which takes the configuration instances in a *std::vector*.

```
std::vector<std::shared_ptr<InjectorConfiguration>> conf;
conf.emplace_back(new Configuration());
conf.emplace_back(new ExtraConfiguration());

auto injector = Injector::create(conf);
```

The alternative create function can be useful for defining the configuration in a single place and using it in the main application and in a configuration validation test. Validation of injector configuration is discussed later in this chapter.

Instances can be obtained directly from the injector via the *getValue<ValueT>* call for non-polymorphic new instances, *getUnique<BaseT>* for polymorphic new instances, *getReference<BaseT>* for polymorphic referenced objects, and *getShared<BaseT>* for polymorphic shared values.

```
auto value  = injector->getValue<MyValueCls>();
auto unique = injector->getUnique<MyBaseCls>();
auto & ref  = injector->getReference<MyReferencedCls>();
auto shared = injector->getShared<MySharedBaseCls>();
```

Alternatively, the unified *getInstance<T>* method may be used to determine which of the four *getValue<ValueT>*, *getUnique<BaseT>*, *getReference<BaseT>*, or *getShared<BaseT>* methods should be called, via compile time examination of the type parameter.

```
// Calls getValue<MyValueCls>()
auto value = injector->getInstance<MyValueCls>();

// Calls getUnique<MyBaseCls>()
auto unique = injector->getInstance<std::unique_ptr<MyBaseCls>>();

// Calls getReference<MyReferencedCls>()
auto & ref = injector->getInstance<MyReferencedCls &>();

// Calls getReference<const MyReferencedCls>()
auto & ref = injector->getInstance<const MyReferencedCls &>();

// Calls getShared<MySharedBaseCls>()
auto shared = injector->getInstance<std::shared_ptr<MySharedBaseCls>>>();

// Calls getShared<const MySharedBaseCls>()
auto shared = injector->getInstance<std::shared_ptr<const MySharedBaseCls>
```

In this documentation, when it is not important to differentiate between the above methods, the expression *get-instance* is used.

Child injectors

Child injectors may be created by calling the *createChild(Conf(), ...)* method. These instance methods take one or more configuration class template parameters and instantiates a child injector with the current injector as the parent.

```
// Create child injector with the specified configuration.
auto c = injector->createChild(ChildConf());
```

The alternative create method is also available for child injector creation. This create method takes the configuration instances in a *std::vector*.

```
std::vector<std::shared_ptr<InjectorConfiguration>> conf;
conf.emplace_back(new ChildConfiguration());
conf.emplace_back(new ExtraChildConfiguration());

auto c = injector->createChild(conf);
```

Alternatively, child injectors may be created by first creating a prototype child injector as above, then repeatedly calling the *createChild(prototype)* method each time a new child injector is required.

```
// Create child injector with the specified configuration.
auto prototype = injector->createChild(ChildConf());

// Create child injector from the prototype.
auto c = injector->createChild(prototype);
```

Using prototype child injectors avoids the build and validation phases of injector construction each time a new child injector is required. The total overhead of creating a child injector from a prototype is limited to the copying of two shared pointers.

It is important to note that the instances of singleton and thread-local singleton bindings of the prototype will be shared between all child injectors created from the prototype. If this is not desired behaviour, then a new child injector must be created via the other *createChild* factory functions that instantiate their own bindings.

Configuration

There are two places where injector configuration is located. The first is within the application's injector configuration. This configuration is the internal wiring of the application. The configuration takes the form of one or more injector configuration classes, instances of which are passed to the injector at construction time. Configuration classes provide the binding information that is used to create the dependency graph.

The second place where injector configuration is located is within participating classes. These are typically each defined with a Balau injector macro in the class declaration. These macros provide information (normally via the *decltype* of direct or indirect class member variables) on the injected dependency types and optionally dependency names that the injector will use during instantiation.

An injector aware class only identifies the injected objects to deliver to the class' injectable constructor, and is independent to the main dependency wiring configuration of a developed software application. Injectable classes may thus be developed independently of the main application, with the application's configuration subsequently wiring them into the dependency graph.

Non-injector aware types and primitive types used for prototype based instance provision do not have any injector macro applied to them. Instances that are constructed manually and passed to a binding call as prototypes, references, or singletons do not require an injector macro, as they are not instantiated by the injector. Similarly, instances that are constructed within a provider function or class that is passed to a binding call do not require an injector macro, as they are not instantiated by the injector either.

Injector configuration

The injector is configured via one or more configuration classes, instances of which are passed to the injector's constructor. An example of an application configuration class is:

```
class Configuration : public ApplicationConfiguration {
    public: void configure() const override {
        bind<Base>().toSingleton<DerivedA>();
        bind<Base2>().toUnique<Derived2>();
    }
};
```

Each configuration class must implement the *configure* method. Binding statements are placed inside application configuration implementation *configure* methods. Environment property name/type declarations are placed inside environment configuration implementation *configure* methods. This documentation chapter discusses application configuration in detail. For more information about environment configuration, see the Environment chapter.

Each binding command in an application configuration class implicates a particular type of binding. In the above example, polymorphic singleton and polymorphic new instance bindings are configured.

Each binding command consists of a two part fluent call. The first *bind* call specifies the interface class as the function template parameter and an optional binding name as a call argument.

The second call specifies the binding type via the function name, along with implementation or provider type as the template parameter for binding types that require one. Bindings that require an object receive the object as a call argument.

The available binding calls are as follows.

Binding type	**Description**
toValue()	Bind a concrete class. A new instance of the class will be stack created each time an instance is requested, and returned via copy elision. The object will be supplied as *ValueT*.
toValue(ValueT)	Bind a prototype value. A new instance of the value will be created each time an instance is requested via copy semantics. The value will be supplied as *ValueT*.

toValueProvider(std::function<ValueT ()>)	Bind a concrete class to a provider function. A new instance of the class will be stack constructed by the provider each time an instance is requested. The object will be supplied as *ValueT*.
toValueProvider<ProviderT>()	Bind a concrete class to an injectable provider class. A new instance of the value class will be stack constructed by the provider each time an instance is requested. The object will be supplied as *ValueT*. The provider will be constructed via standard injection of the provider's dependencies.
toValueProvider<ProviderT>(std::shared_ptr<ProviderT>)	Bind a concrete class to an injectable provider instance. The provider instance is supplied in a shared pointer container, allowing the caller to retain shared ownership if required. A new instance of the value class will be stack constructed by the provider each time an instance is requested. The object will be supplied as *ValueT*. The provider will be constructed via standard injection of the provider's dependencies.
toUnique<DerivedT>()	Bind an interface to an implementing class. A new instance of the class will be heap constructed each time an instance is requested. The object will be supplied as *std::unique_ptr<BaseT>*.
toUniqueProvider(std::function<std::unique_ptr<BaseT> ()>)	Bind an interface to a provider function. A new instance deriving from the base type will be heap constructed by the provider each time an instance is requested. The object will be supplied as *std::unique_ptr<BaseT>*.
toUniqueProvider<ProviderT>()	Bind an interface to an injectable provider class. A new instance deriving from the base type will be heap constructed by the provider each time an instance is requested. The object will be supplied as *std::unique_ptr<BaseT>*. The provider will be constructed via standard injection of the provider's dependencies.

toUniqueProvider<ProviderT>(std::shared_ptr<ProviderT>)	Bind an interface to an injectable provider class. The provider instance is supplied in a shared pointer container, allowing the caller to retain shared ownership if required. A new instance deriving from the base type will be heap constructed by the provider each time an instance is requested. The object will be supplied as *std::unique_ptr<BaseT>*. The provider will be constructed via standard injection of the provider's dependencies.
toReference(BaseT &)	Bind a reference type to the supplied reference object. A reference to the object referenced in the injector's configuration will be returned on each call. The object will be supplied as *BaseT &*.
toThreadLocal<DerivedT>()	Bind an interface to an implementing class with thread-local singleton semantics. The thread-local singleton will be heap constructed lazily for each new calling thread. The object will be supplied as *std::shared_ptr<BaseT>*.
toThreadLocal()	Bind a concrete class with thread-local singleton semantics. The thread-local singleton will be heap constructed lazily for each new calling thread. The object will be supplied as *std::shared_ptr<T>*.
toSingleton<DerivedT>()	Bind an interface to an implementing class with singleton semantics. The singleton will be heap constructed lazily. The object will be supplied as *std::shared_ptr<BaseT>*.
toSingleton()	Bind a concrete class with singleton semantics. The singleton will be heap constructed lazily. The object will be supplied as *std::shared_ptr<T>*.
toSingleton(std::shared_ptr<BaseT>)	Bind an interface to the supplied singleton object. The injector will share ownership of the pointer. The object will be supplied as *std::shared_ptr<BaseT>*.
toSingleton(BaseT *)	Bind an interface to the supplied singleton object. The injector will take ownership of the pointer. The object will be supplied as *std::shared_ptr<BaseT>*.

toSingletonProvider()	Bind a singleton provider class for singleton semantics. The singleton will be provided by instantiating the provider and calling it a single time during injector creation. The object will be supplied as *std::shared_ptr<T>*.
toSingletonProvider(std::shared<ProviderT>)	Bind a singleton provider instance for singleton semantics. The singleton will be provided by calling the provider a single time during injector creation. The provider will then be dereferenced. The object will be supplied as *std::shared_ptr<T>*.
toEagerSingleton<DerivedT>()	Bind an interface to an implementing class with singleton semantics. The singleton will be heap constructed eagerly. The object will be supplied as *std::shared_ptr<BaseT>*.
toEagerSingleton()	Bind a concrete singleton class. The singleton will be heap constructed eagerly. The object will be supplied as *std::shared_ptr<T>*.

Each of the binding types belongs to one of four meta-types. These meta-types are:

- Value;
- Unique;
- Reference;
- Shared.

These classifications correspond to the four types of instance provided by the injector.

Two of the four meta-type classifications are also available in const form. There are thus effectively six meta-types in total:

- Value;
- Unique;
- Reference;
- Const Reference;
- Shared;
- Const Shared.

The meta-type classification and const qualifier of a binding forms part of the binding key (the other parts being the typeid and the name). The six meta-type classifications thus form six binding groups. In each binding group, the typeid and name must be unique. If an attempt is made to create an injector with a configuration that has duplicate typeid/name pairs in a classification group, a *DuplicateBindingException* will be thrown during creation of the injector.

When the injector completes the configuration phase, a validation phase is run. This validation phase verifies that all the dependencies required by each injectable participating class (i.e. a class that takes one or more dependencies) can be satisfied by the injector. If this is not the case, the injector throws a *NoBinding* exception.

The validation phase ensures that any binding issues that may occur with registered classes are caught at the time of injector instantiation. The only subsequent binding runtime errors that may occur are thus direct attempts to obtain instances from the injector for bindings that do not exist.

Reference bindings

Although reference bindings are conceptually simple, care must be taken with regard to referenced object lifetimes.

As referenced objects are not instantiated inside the injector nor does the injector take ownership of the object, the injector has no control on the lifetime of the supplied objects. Consequently, it is important to take into account that the lifecycle management of referenced objects is the responsibility of the application and not of the injector.

The policy for referenced object lifetimes must therefore be to ensure that all objects passed to the injector's configuration for referencing remain alive past the end of the injector's lifetime and the lifetimes of other objects that have obtained references to these objects from the injector.

Due to the potential complexity of managing this, it is best to limit the use of reference bindings to that of objects that are easily known to have sufficiently long lives.

Const bindings

The injector supports *const* bindings for *Reference* and *Shared* binding types.

If *const* bindings are specified for for *Value* or *Unique* binding types, the *const* qualifier will be stripped from the type and a warning logged to the *balau.injector*. Stripping the *const* qualifier from *Value* or *Unique* binding types will not affect the injector semantics, as bindings of these meta-types produce new instances. If a new instance needs to be const, the instance can be set to const at the calling site.

Injectable classes

Base interfaces / abstract base classes do not require an injector macro and are no different from any other abstract C++ class.

Concrete implementation classes that are to be instantiated by the injector require an injector configuration macro. This macro specifies the types (via *decltype* for the *BalauInjectConstruct* / *BalauInjectConstructNamed* and *BalauInject* / *BalauInjectNamed* macros, and directly for the *BalauInjectTypes* / *BalauInjectNamedTypes* macros) and optional names of the injected dependencies. The *BalauInjectConstruct* / *BalauInjectConstructNamed* macros also define an injectable constructor.

Injector macros are currently defined with up to sixteen unnamed or named dependencies.

Inject-construct macros

The following is an example of an injectable class that uses an inject-construct macro.

```
class Derived2 : public Base2 {
    // The dependency that is populated via the constructor.
    private: std::shared_ptr<Base> dependency;

    // The injector boilerplate.
    BalauInjectConstruct(Derived2, dependency)

    public: void foo2() override;
};
```

The macro *BalauInjectConstruct* specifies that this implementation of *Base2* takes a single dependency. The specified member variable's type will be used to form the factory method in the class and the corresponding constructor.

The resulting implicit constructor defined by the *BalauInjectConstruct* macro is as follows.

```
private: explicit Derived2(std::shared_ptr<Base> dependency_)
    : dependency(std::forward<std::shared_ptr<Base>>(dependency_)) {}
```

The automatically generated constructor will move value, unique pointer, and shared pointer *rvalue* type dependencies into the member variables, and will assign supplied reference *lvalue* type dependencies to their associated class members.

The general form of the automatically generated constructors via the *BalauInjectConstruct* / *BalauInjectConstructNamed* macros is as follows (d0, d1, d2, ... are the member variables /references of the class).

```
private: ClassName(decltype(d0) d0_,
                   decltype(d1) d1_,
                   decltype(d2) d2_,
                   // ... up to 16 in total ...
) : d0(std::forward<decltype(d0)>(d0_))
  , d1(std::forward<decltype(d1)>(d1_))
  , d2(std::forward<decltype(d2)>(d2_))
    // ... up to 16 in total ...
{}
```

The two possible formats of the macro are:

```
BalauInjectConstruct(ClassName, MemberVariable ... )
BalauInjectConstructNamed(ClassName, { MemberVariable, Name } ... )
```

where:

- *ClassName* is the name of the class;
- *X* is the number of dependencies that the implicitly defined class constructor will take;
- *MemberVariable* is the direct or indirect member variable to be set in the constructor;
- *Name* is the name of the dependency.

The first parameter in the macros is always the name of the class. As C++ does not have any way of specifying the class' type within a class declaration, the class name must be provided to the macro.

The *MemberVariable* entries specified in the macro are used in *decltype* expressions in order to obtain the required types for the dependencies.

Inject only macros

In addition to the *BalauInjectConstruct* / *BalauInjectConstructNamed* macros, a similar pair of *BalauInject* / *BalauInjectNamed* macros exist.

```
BalauInject(ClassName, MemberVariable ... )
BalauInjectNamed(ClassName, { MemberVariable, Name } ... )
```

These macros are identical to the *BalauInjectConstruct* / *BalauInjectConstructNamed* macros with the exception that they leave the definition of the injectable constructor to the end developer. This allows the injectable constructor to be customised. The notable requirement for this is when injected objects are consumed in a non-uniform way.

The following is an example of a constructor-less macro used to specify named dependencies.

```
class Derived2WithNamed : public Base2 {
    private: std::shared_ptr<Base> dependency;

    // The injector boilerplate for a named dependency.
    BalauInjectNamed(Derived2WithNamed, dependency, "namedBase")

    // Explicitly defined injectable class, allowing customisation.
    private: explicit Derived2WithNamed(std::shared_ptr<Base> aDependency)
        : dependency(std::move(aDependency)) {
        capture.add("Derived2WithNamed constructor");
    }

    public: ~Derived2WithNamed() override = default;

    public: void foo2() override {
        capture.add("Derived2WithNamed.foo2");
        dependency->foo();
    }
};
```

One notable use case for using the *BalauInject* / *BalauInjectNamed* macros instead of the *BalauInjectConstruct* / *BalauInjectConstructNamed* macros is when indirect member variables need to be specified.

Another example of the use of an explicit injectable constructor can be seen in the *HttpServer* class of the Balau library. This example illustrates the use of indirect member variables to define the injected types. Several of the injected objects are consumed by the inner *state* object instead of direct fields, necessitating an explicit definition of the injected constructor.

```
// The injector macro.
BalauInjectNamed(
      HttpServer
    , state->injector,    ""
    , state->serverId,    "httpServerIdentification"
    , state->endpoint,    "httpServerEndpoint"
    , threadNamePrefix,   "httpServerThreadName"
    , workerCount,        "httpServerWorkerCount"
    , state->httpHandler, "httpHandler"
    , state->wsHandler,   "webSocketHandler"
    , state->mimeTypes,   "mimeTypes"
);

// The explicitly defined injectable constructor.
HttpServer(std::shared_ptr<Injector> injector,
           std::string serverIdentification,
           TCP::endpoint endpoint,
           std::string threadNamePrefix_,
           size_t workerCount_,
           std::shared_ptr<HttpWebApp> httpHandler,
           std::shared_ptr<WsWebApp> wsHandler,
           std::shared_ptr<MimeTypes> mimeTypes);
```

Inject types macros

The standard injector macros use the direct or indirect field names of the class in *decltype* expressions in order to obtain the required types for the dependencies. This approach is compact and efficient and should be used in the majority of cases. However, these macros will not work if:

- one or more of the dependencies should be specified as different but compatible types to the ones derived via *decltype*, (an example of which is promoting a *std::unique_ptr<BaseT>* to a *std::shared_ptr<BaseT>* member variable in the constructor initialisation);
- one or more of the dependencies are used without assigning them to direct or indirect member variables.

The alternative *BalauInjectTypes* / *BalauInjectNamedTypes* macros perform a similar job to the standard macros, but take the types of the dependencies instead of the direct or indirect member variable names.

```
BalauInjectTypes(ClassName, DependencyType ... )
BalauInjectNamedTypes(ClassName, { DependencyType, Name } ... )
```

If the *HttpServer* class were to be declared with a *BalauInjectNamedTypes* macro instead of the *BalauInjectNamed* macro, the source code would look like the following extract.

```
// The injector macro with explicit dependency type information.
BalauInjectNamedTypes(
      HttpServer
    , std::shared_ptr<Injector>,   ""
    , std::string,                 "httpServerIdentification"
    , TCP::Endpoint,               "httpServerEndpoint"
    , std::string,                 "httpServerThreadName"
    , size_t,                      "httpServerWorkerCount"
    , std::shared_ptr<HttpWebApp>, "httpHandler"
    , std::shared_ptr<WsWebApp>,   "webSocketHandler"
    , std::shared_ptr<MimeTypes>,  "mimeTypes"
);

// The explicitly defined injectable constructor.
HttpServer(std::shared_ptr<Injector> injector,
           std::string serverIdentification,
           TCP::endpoint endpoint,
           std::string threadNamePrefix_,
           size_t workerCount_,
           std::shared_ptr<HttpWebApp> httpHandler,
           std::shared_ptr<WsWebApp> wsHandler,
           std::shared_ptr<MimeTypes> mimeTypes);
```

Given that the types of the direct or indirect member variables of an injectable class often match the types of the dependencies, use of the *BalauInjectTypesX* macros should only occur in a minority of cases.

Instantiation

Injector

Once one or more suitable configuration classes have been defined, an injector instance may be created by calling the *Injector::create(conf, ...)* function:

```
std::shared_ptr<Injector> injector = Injector::create(Config1(), Config2()
```

This instantiates an injector instance in a *std::shared_ptr<Injector>* and initialises the bindings from the supplied configuration(s). The *auto* keyword can be used to condense the statement:

```
auto injector = Injector::create(Config1(), Config2());
```

Injectors can only be instantiated within a *std::shared_ptr<Injector>*. This allows them to supply themselves as a dependency when required (via *shared_from_this()*), and also be accessed as class member fields of type *Injector* in classes that require direct access to the injector.

An injector may be shared throughout the application by copying the shared pointer. Injectors may be used across multiple threads of the application without any synchronisation.

Instances

In order to obtain instances from an injector, four templated method calls are available:

```
ValueT                 stackInstance = injector.getValue<ValueT>();
std::unique_ptr<BaseT> heapInstance  = injector.getUnique<BaseT>();
BaseT &                reference     = injector.getReference<BaseT>();
std::shared_ptr<BaseT> singleton     = injector.getShared<BaseT>();
```

The type *BaseT* can be non-const or const for reference and shared bindings.

Depending on the injector's configuration:

- *getValue* calls may access instance objects created on the stack and moved, or copy constructed from a prototype object;
- *getUnique* calls may obtain polymorphic instance objects created on the heap;
- *getReference* calls may access long lived objects referenced from within the configuration used to construct the injector;
- *getShared* calls may access singletons or thread-local singletons created on the heap.

In addition to the above calls, there is a unified templated method call that resolves the meta-type by specialising on the supplied type parameter.

```
T object = injector.getInstance<T>();
```

Unlike the four previous template functions that all accept the direct value or base type of the instance(s) represented by the binding, the *getInstance* template method resolves at compile time to:

- *getShared<T>* if the specified type is *std::shared_ptr<T>*;
- *getReference<T>* if the specified type is *T &*;
- *getUnique<T>* if the specified type is *std::unique_ptr<T>*;
- *getValue<T>* otherwise.

The *getInstance* template method is useful when an injector is used within a template class or function, where the exact type to be requested is deduced by the compiler.

Const bindings

Bindings may be created *const* for reference and shared meta-types. For example, the following application creates an injector configuration with a const reference binding and a const singleton binding, along with an injected double value, then gets the objects from the constructed injector.

```
#include <Balau/Application/Injector.hpp>

struct A {
    double value;

    BalauInjectConstruct(A, value);

    A(A &) = delete; // Prevent copying.
};

const A a(543.2);

int main () {
    class Configuration : public ApplicationConfiguration {
        public: void configure() const override {
            // A double value injected into A.
            bind<double>().toValue(123.456);

            // Bind a const reference.
            bind<const A>().toReference(a);

            // Bind a const singleton.
            bind<const A>().toSingleton();
        }
    };

    auto injector = Injector::create<Configuration>();

    auto & r = injector->getReference<const A>();
    auto a = injector->getShared<const A>();
}
```

The injector calls return a const reference and a shared pointer containing a const pointer. Note that the reference call requires an ampersand after the *auto* type keyword in order for the code to compile.

Care should be take with regard to getting references from the injector. If the copy constructor of class *A* were not deleted, the code would compile if the ampersand were removed.

```
// Class A2 has a copy constructor..
auto a3 = injector->getReference<const A2>();
```

The result of this would be a copy of the reference instead of a reference to it. Such semantics are best created via the *toValue(prototype)* binding call instead.

Consequently, it is wise to delete the copy constructor of classes that are destined to be referenced via the injector. This will enforce referencing at compile time.

Const promotion

When a get-instance call is made for a const object, the resulting binding used may be a non-const binding that is *promoted* to a const binding. This non-const to const binding semantics of the injector parallels the non-const to const binding semantics of the C++ language.

Although value and unique meta-types do not support const bindings, const promotion nevertheless applies to these non-polymorphic and polymorphic new instance binding types. These const promotions are similar to that in C++ when a non-const object is copy assigned to a new const declared object.

When the injector is part of a hierarchy, const promotion applies to the whole hierarchy. The whole hierarchy will thus first be checked for a const binding, then the whole hierarchy will be checked again for a non-const binding.

The promotion rules are listed in the following table. The left hand column lists the requested const types. The middle column lists the default binding type supplied by the injector if a binding of that type is available. If such a binding is not available, the injector will lookup a binding of the type listed in the third column.

Requested meta-type	**Default provided meta-type**	**Promoted provided meta-type**
const ValueT	-	*ValueT*
std::unique_ptr<const BaseT>	-	*std::unique_ptr<BaseT>*
const std::unique_ptr<const BaseT>	-	*std::unique_ptr<BaseT>*
const BaseT &	*const BaseT &*	*BaseT &*
std::shared_ptr<const BaseT>	*std::shared_ptr<const BaseT>*	*std::shared_ptr<BaseT>*
const std::shared_ptr<const BaseT>	*std::shared_ptr<const BaseT>*	*std::shared_ptr<BaseT>*

Weak promotion

When a get-instance call is made for a *std::weak_ptr<BaseT>*, the binding request will be *promoted* to a shared binding. Weak pointer fields are thus initialised via shared bindings during injection.

The promotion rules for weak pointers are as follows.

Requested meta-type	**Default provided meta-type**	**Fallback provided meta-type**
std::weak_ptr<BaseT>	*std::shared_ptr<BaseT>*	-
std::weak_ptr<const BaseT>	*std::shared_ptr<const BaseT>*	*std::shared_ptr<BaseT>*
const std::weak_ptr<const BaseT>	*std::shared_ptr<const BaseT>*	*std::shared_ptr<BaseT>*

Custom deleters

The C++ *std::unique_ptr* and *std::shared_ptr* containers can be created with custom deletion policies. These allow deletion of pointers at the ends of their lifespans via deletion mechanisms other than the standard *delete* call as provided by *std::default_delete*.

The mechanism by which custom deletion polices is specified in C++ is different for each pointer container type. The Balau injector thus provides two different mechanism for deletion policy specification in binding calls, one for *std::unique_ptr* and another for *std::shared_ptr*. Accordingly, the mechanism for obtaining unique and shared instances that have custom deleters is different for each binding type.

Unique custom deletion

The C++ *std::unique_ptr* container requires a custom deletion policy to be specified as a type argument to the unique pointer class template. This is thus performed at compile time, and becomes part of the pointer container's type.

Due to this, the deleter type of a *unique* binding is part of the binding key. For *unique* bindings that do not have a custom deletion policy, the binding key deleter type is *std:: default_delete<BaseT>*.

In order to specify a custom deleter for a unique binding, a custom deleter type is specified in the first part of the fluent call chain, i.e. as a type argument to the *bind()* call.

```
//
// A custom deleter.
//
struct CustomDeleter {
    public: void operator () (U * object) {
        log.trace("Object deleted {}", (size_t) object);
        delete object;
    }
};

//
// Custom deleter type specified for a binding.
//
class Configuration : public ApplicationConfiguration {
    public: void configure() const override {
        // A unique binding for U, with std::default_delete<BaseT>.
        bind<U>().toUnique<V>();

        // A unique binding for U, with custom deleter type CustomDeleter.
        bind<U, CustomDeleter>().toUnique<V>();
    }
};
```

For other binding types, any deleter type specified in the *bind()* call will be ignored.

As the custom deleter type is part of the binding key for unique bindings, it must be specified in order to obtain a polymorphic new instance of the specified type with the custom deletion policy.

```
// Create an injector with the above configuration.
auto injector = Injector::create(Configuration());

// Get a polymorphic new instance specified by binding {U, std::default_de
auto a = injector->getUnique<U>();

// Get a polymorphic new instance specified by binding {U, CustomDeleter}.
auto b = injector->getUnique<U, CustomDeleter>();
```

Shared custom deletion

The C++ *std::shared_ptr* container requires a custom deletion policy to be specified as an argument to the shared pointer's constructor. Although custom deleter types for shared bindings are specified at compile time in the Balau binding configuration fluent call chain, the deleter instance itself is supplied at runtime to the C++ *std::shared_ptr* container. The deleter type does not thus become part of the pointer container's type.

Due to this, the deleter type of a *shared* binding is not part of the binding key.

In order to specify a custom deleter for a shared binding, a custom deleter type is specified in the second part of the fluent call chain, i.e. as a type argument to the *toSingleton()*, *toEagerSingleton()*, and *toThreadLocal()* calls.

```
// Custom deleter type specified for a binding.
class Configuration : public ApplicationConfiguration {
    public: void configure() const override {
        //
        // A shared binding for U, with std::default_delete<BaseT>.
        //
        bind<U>().toShared<V>();

        //
        // A shared binding for U, with custom deleter type CustomDeleter.
        //
        // A name is required, otherwise the binding would be identical
        // to the previous one.
        //
        bind<U>("custom").toShared<V, CustomDeleter>();
    }
};
```

As the custom deleter type is not part of the binding key for shared bindings, it must not be specified in order to obtain a polymorphic shared instance of the specified type, regardless of whether or not the shared instance has a custom deletion policy.

```
// Create an injector with the above configuration.
auto injector = Injector::create(Configuration());

// Get a polymorphic shared instance specified by binding {U, ""}.
auto a = injector->getShared<U>();

// Get a polymorphic shared instance specified by binding {U, "custom"}.
auto b = injector->getShared<U>("custom");
```

Injector hierarchies

The injector resolves instances from its binding configuration or from its parent injector. Injectors can form a hierarchy, the binding configurations of which are queried in turn when an instance is requested.

Child injector creation

In order to construct a child injector, the *createChild(Conf(), ...)* member function is used.

```
auto childInjector = parent->createChild(Config());
```

This method call is identical to the function used to create a parentless injector, with the exception that it is a member function.

Prototype child injectors

Child injectors may also be created by first creating a prototype child injector as discussed previously, then repeatedly calling the *createChild(prototype)* method each time a new child injector is required.

```
// Create child injector with the specified configuration.
auto prototype = injector->createChild(ChildConf());

// Create child injector from the prototype.
auto c = injector->createChild(prototype);
```

Using prototype child injectors avoids the build and validation phases of injector construction each time a new child injector is required. The total overhead of creating a child injector from a prototype is thus limited to the copying of two shared pointers.

It is important to note that the instances of singleton and thread-local singleton bindings of the prototype will be shared between all child injectors created from the prototype. If this is not desired behaviour, then a new child injector must be created via the other *createChild* functions that instantiate their own bindings.

Injector callbacks

An additional feature of the injector is the ability to register post-construction and pre-destruction callbacks. Registered callbacks will then be called by the injector, either directly after injector creation (for post-construction callbacks), or immediately before injector destruction (for pre-destruction callbacks).

Standard callbacks

Registering callbacks provides a convenient way to execute program logic immediate after injector creation and/or immediately before injector destruction. Post-construction and pre-destruction callbacks are also useful for the explicit management of cyclic dependencies between singletons (discussed in the next section).

The only restriction to the program logic that may be run within a callback is that pre-destruction callbacks must be *noexcept(true)*. This is because the pre-destruction callbacks are run from within the injector destructor.

The signatures of the callback registration methods are as follows.

```
// Post-construction callback registration.
void registerPostConstructionCall(const std::function<void (const Injector

// Pre-destruction callback registration.
void registerPreDestructionCall(const std::function<void ()> & call) const
```

The signatures of the callbacks are thus:

```
// Post-construction function signature.
const std::function<void (const Injector &)>

// Pre-destruction function signature.
const std::function<void ()>
```

Post-construct callbacks are supplied with a reference to the injector. Pre-destruction callbacks are not. Although pre-destruction callbacks must be *noexcept(true)*, the pre-destruction function signature does not contain *noexcept(true)*, as this is not yet handled by *std::function* in C++17. Despite this, functions registered as pre-destruction callbacks must nevertheless be *noexcept(true)*.

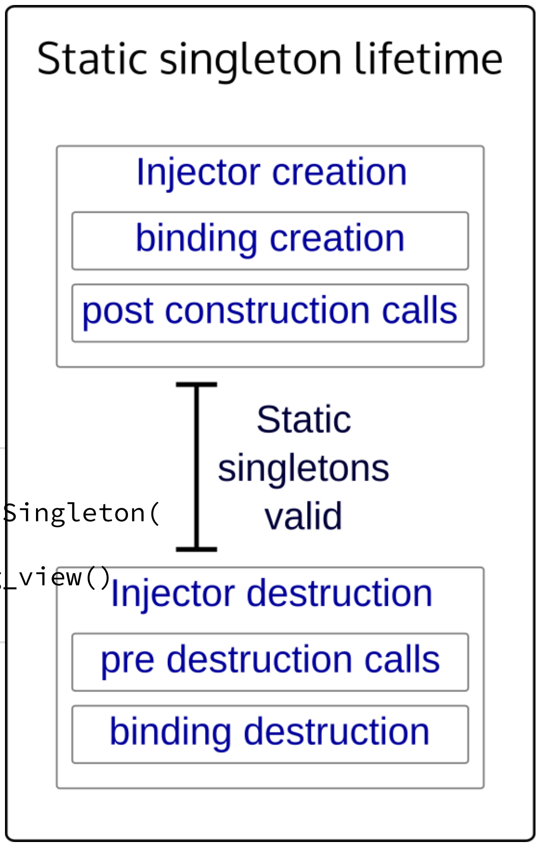
Singleton callback

The static singleton call convenience method provides registration of shared pointer containers that will be managed by the injector post-construction and pre-destruction. This method allows a singleton to be statically available in the application, between the post-construction and pre-destruction execution points.

The signature of the static singleton registration method is as follows.

```
// Static singleton pointer registration.
template <typename T> void registerStaticSingleton(
      std::shared_ptr<T> * ptrPtr
    , std::string_view name = std::string_view()
) const;
```

The diagram on the right illustrates the region of validity for static singleton pointers registered via the *registerStaticSingleton* injector method. The execution time-line travels from top to bottom. Static singleton pointers are not guaranteed to be



valid during binding creation or binding destruction. Dereferencing them directly or indirectly from within the constructors of other injectables may thus result in segmentation faults, depending on the non-deterministic order of the construction of singletons.

In order to ensure that the static singleton registrations methods are called during inject construction, singleton bindings containing static singleton registration calls must be eager, or the singletons must be dependencies of eager singletons. Otherwise, the singletons may not be constructed during binding construction and the registration callbacks will never be executed.

Static singleton registration is a feature that is aimed solely for developers that are rearchitecting a codebase with hard-wired singletons to one that uses dependency injection. Use of static singleton registration is not recommended for greenfield projects. Migration away from static singleton registration should also be planned for rearchitected codebases that have been moved to a dependency injection architecture.

Cyclic dependencies

This section discusses how automatic cyclic dependencies are prevented by the injector and how to manually manage cyclic dependencies between instances.

Configuration cycles

The Balau injector provides a constructor injection paradigm. One consequence of this is that if a cyclic dependency is created between instances obtained from the injector, the application will crash due to a call stack overflow for stack based types or a segmentation fault for heap based types.

Due to this, the injector runs cyclic dependency analysis in the validation phase run during injector instantiation. If a cycle is found in the binding dependency tree constructed from the supplied configuration, a *CyclicDependencyException* is thrown.

Explicitly managed cycles

If a cyclic relationship between two instances is required, this must be managed explicitly by the application. If an explicitly constructed cyclic relationship approach is used using shared pointer containers, the normal rules in C++ regarding *std::shared_ptr* cycles apply and must be managed accordingly.

The best way to achieve an explicitly managed cyclic dependency is by creating a weak or shared pointer in one of the cyclically dependent classes, then registering a post-construction callback with the injector from within the constructor of the class. This callback can then set the pointer container to point to the other instance, by obtaining the instance from the injector supplied in the callback.

The choice of weak or shared pointer will depend on the chosen destruction strategy. If no action is taken to explicitly remove the cyclic relationship, then a weak pointer should be used. This ensures that there is no permanent cyclic dependency in place. The inconvenience with using a weak pointer is that the pointer must be obtained via the *lock* call each time the pointer is required.

If a shared pointer is used, then a pre-destruction callback should be registered with the injector from within the constructor of one of the instances that form the cyclic dependency (typically the same class that contains the post-construction callback registration call). This pre-destruction callback should reset the shared pointer, breaking the cyclic dependency before the injector destructor destroys the binding map.

In order to perform explicit management of cyclic pointers, the injector needs to be injected into one of the cyclically dependent classes.

Injecting the injector

In order to inject the injector, it is sufficient to specify the injector type as a weak pointer, either via an injector macro that specifies the relevant injector field in the class (*BalauInject*, *BalauInjectConstruct*, *BalauInjectNamed*, or *BalauInjectConstructNamed*), or explicitly via an injector macro that specifies the exact type to be injected (*BalauInjectTypes*, or *BalauInjectNamedTypes*). Once this is done, the injector will inject itself during a get-instance call.

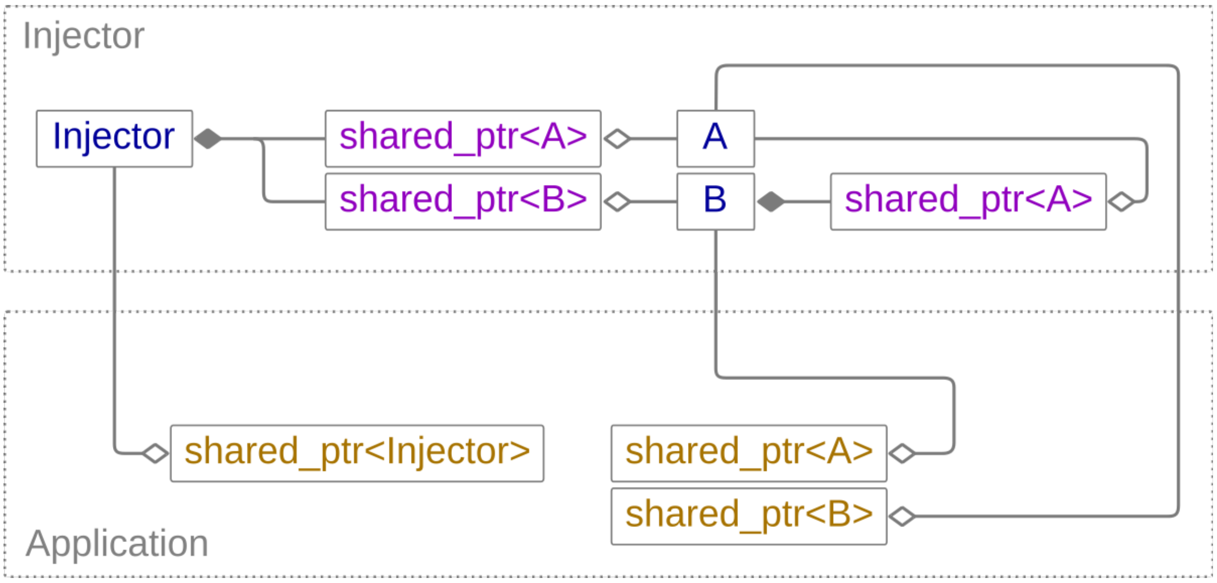
As singleton instances may be created during the creation of the injector, it is important to note that such singletons must not use the injector from within their constructor. Instead, a pointer to the injector should be maintained within a field of the class and set in the constructor initialisation list. The injector can then be used in non-constructor methods in the class.

If the injector is nevertheless required during construction, a post construct callback may be registered with the injected injector. This callback will be executed by the injector immediate after construction is complete.

Injection of the injector can only be achieved via a *std::weak_ptr<Injector>*, either specified as a field of the injectable class, or explicitly via the injector macros that specify the exact types of the dependencies. This ensures that the writer of the injectable class be aware that maintaining a shared pointer to the injector in the instance will result in a cyclic dependency if the instance is a singleton (i.e. owned by the injector). If a shared pointer is used to reference the injected injector, the injector will become a node within an implicitly created dependency cycle.

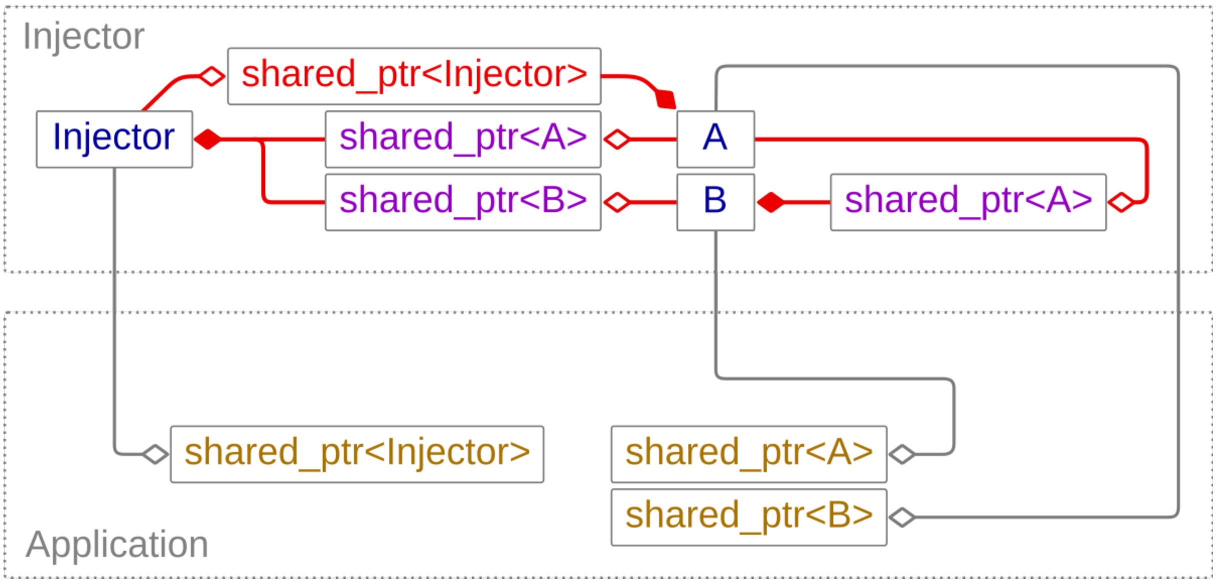
Injector cycles

As previously mentioned, injection of the injector can only be performed if the receiving type is a *std::weak_ptr<Injector>*. An example of what would happen if a shared pointer is used instead is illustrated in the following diagrams. The diagram below shows a set of relationships between an injector, two singleton bindings, and some shared pointers obtained from the injector by the application. Instances are shown in blue, shared pointers internal to the instances are shown in purple, and shared pointers in the application code are shown in brown.



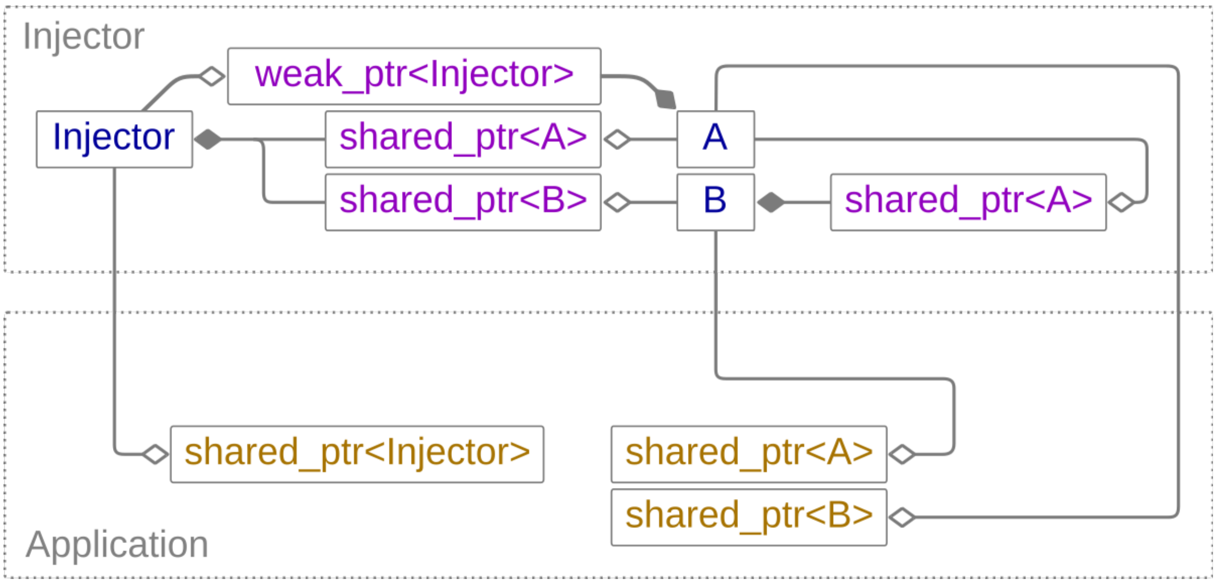
In this example, singleton instance *B* has a shared pointer field to singleton instance *A*. It can be verified that there are no cycles by following sequences of aggregation/composition paths and nodes. All paths lead to the terminal node *A*.

If a *std::shared_ptr<Injector>* member variable is added to *A*, then the relationships change to those in the diagram below. Two cycles have been formed within the pathways. The new *std::shared_ptr<Injector>* member variable of *A* is shown in red. Also shown in red are the path segments that form the cycles.



If an attempt to instantiate an injector is made with such a configuration, the injector will throw a *SharedInjectorException* during the validation phase.

The solution to this is to use a weak pointer when a *Shared* binding needs the injector to be injected into it. If a *std::weak_ptr<Injector>* member variable is created in *A* instead of the previous *std::shared_ptr<Injector>*, no such exception is thrown by the injector. The relationships created by this modified configuration are illustrated below.



The *std::weak_ptr<Injector>* forms a dependency break in the pathways and consequently there are no cycles present in the dependency graph.

Regardless of the above, it is important to note that the normal rules of C++ still apply after the weak pointer is supplied. If an injector shared pointer is manually created in the injectable class' constructor from a supplied injector weak pointer and then subsequently used to set a shared pointer field, a cyclic dependency will be created and the injector will never be destroyed. Note that using a pre-destructor callback will not work either, as these callbacks are run from within the injector's destructor, which will never get called. It is thus essentially up the end developer to respect the requirement that pointers to the injector in injectable classes be maintained as weak pointers.

Configuration testing

Once the configuration(s) of the application's injector(s) have been created, they can be unit tested via one or more simple unit tests.

The injector provides two static methods *validate* and *validateChild*. The first method *validate* validates root injector configurations and the second method *validateChild* validates child configurations.

Root injectors

Unit testing an injector configuration that does not contain any eager singletons could be achieved by simply instantiating the injector. However, if the configuration has any eager singletons, they would be instantiated in the constructor. The *validate* method thus performs injector instantiation without eager singleton instantiation.

The unit test can consist of a single statement. The test passes if no exception is thrown.

```
// Test the main runtime configuration of the application.
void InjectorConfigurationTest::mainRuntimeConfiguration() {
    // Configuration objects obtained at runtime.
    Config1 config1();
    Config2 config2();

    Injector::validate(config1, config2);
}
```

The vector based validate function is also available if required.

```
// Test the main runtime configuration of the application.
void InjectorConfigurationTest::mainRuntimeConfiguration() {
    std::vector<std::shared_ptr<InjectorConfiguration>> conf;
    conf.emplace_back(new Config1());
    conf.emplace_back(new Config2());

    Injector::validate(conf);
}
```

The vector based validate function is useful because the application's final configuration can be defined in a single place, and then accessed by the main application and the validation unit test. No duplication of the configuration instantiation list is then necessary, and the unit test automatically picks up configuration instance changes without the test needing any modification.

Child injectors

Validation of child injector configuration requires a suitable parent injector to be supplied to the validation method. In order to avoid the eager singleton issues discussed previously, the root injector *validate* method returns a *ValidationParent* object that represents the validated parent injector for use in child injector validation calls.

```
// Test the child configuration.
void InjectorConfigurationTest::childConfiguration() {
    auto parent = Injector::validate(Config1(), Config2());
    Injector::validateChild(parent, Config3());
}
```

The *validateChild* method also returns a *ValidationParent* object, allowing deep injector hierarchies to be validated.

```
// Test 4 levels of child configuration.
void InjectorConfigurationTest::childConfiguration() {
    auto parent1 = Injector::validate(Config1(), Config2());
    auto parent2 = Injector::validateChild(parent1, Config3());
    auto parent3 = Injector::validateChild(parent2, Config4());
    Injector::validateChild(parent3, Config5(), Config6());
}
```

The alternative vector based *validateChild* method is also available if required for child injector testing.

Logging

At creation time, the injector logs the dependencies to the "balau.injector" logging namespace, and the dependency tree to the "balau.container" logging namespace. This logging can be useful for debugging dependency issues.

The logging output is set to TRACE level. In order to see one or both of these logging outputs, set the "balau.injector" and/or "balau.container" logging namespaces to log at TRACE level.

Design

This section provides a summary of some aspects of the philosophy and design of the Balau injection framework. It is not necessary to read this section in order to use the injector.

Overview

The design of the Balau injection framework was partly influenced from experience with the Java and C# based *Guice*, *Spring*, and *Unity* dependency injection frameworks in enterprise software development. The Balau injector has a technical approach that reflects the more comprehensive type system available in the C++ language and standard library, and the runtime reflection limitations in C++.

Background

Java based dependency injection frameworks work within the confines of the Java type system. The combination of dual primitive/reference types and generics type erasure has resulted in Java based injectors having an API based on a single meta-type: the Java reference.

Outside of the compiler imposed *final* keyword, Java references may be bound and rebound. Assigning a Java reference copies the reference "value", resulting in a new reference "value" that points to the same object. In the context of C++, the Java reference is most similar to a C++ pointer. Java dependency injection frameworks thus effectively work wholly with pointers to objects.

In C++, we have a much richer type system than in Java. We also have a responsibility for managing object lifetime that can only be partially automated via pointer containers.

Given the richer type system, a C++ dependency injection framework does not need to be limited to providing *T ** pointers to objects. Potentially some or all or more than the following meta-types could be supplied.

Meta-type	Description
T	value
const T	const value
*T **	pointer to object
*const T **	pointer to const object
*T * const*	const pointer to object
*const T * const*	const pointer to const object
T &	reference to object (*final* pointer)
const T &	reference to const object
unique_ptr<T>	uniquely owned pointer
const unique_ptr<T>	const uniquely owned pointer to object
unique_ptr<const T>	uniquely owned pointer to const object
const unique_ptr<const T>	const uniquely owned pointer to const object
shared_ptr<T>	shared ownership pointer
const shared_ptr<T>	const shared ownership pointer to object
shared_ptr<const T>	shared ownership pointer to const object
const shared_ptr<const T>	const shared ownership pointer to const object

Meta-design

The key questions raised during the development of the Balau injection framework were:

1. which meta-types should be provided by the injector;
2. how much of the work should be done at compile time;
3. how should the lack of Java-like annotations be mitigated;
4. behind what kind of API should this be encapsulated?

Some of the key requirements determined during the injector design phase were:

1. the injector must be a simple, standard (non-template) class *Injector*, able to be trivially used in a class member variable declarations;
2. magic injection (injection of types or provision of instances not registered with the injector) must not be supported;
3. the API must be simple to use;
4. the injection framework code must be simple to debug when an end developer is faced with a failing binding or a non-obvious injection issue;
5. the injector must be able to inject itself;

6. injector creation must fail if there are dependency tree issues or cyclic dependencies;
7. application configuration must be represented in C++ code (i.e. no DSL or XML files);
8. environment configuration must be represented by text based property files, loaded into the injector during instantiation.

It rapidly became clear that the technical implications of these questions and requirements were tightly coupled. Allowing a lot of technical freedom in the technical solution to one of the questions often resulted in unacceptable limitations for the technical solutions to one or more of the other questions and requirements.

The final chosen design aims to reflect the common needs of enterprise software development with a simple API, whilst maintaining safety, testability, and minimising feature shrinkage. Good performance was a requirement, but not to the point of detriment to other requirements. Enterprise dependency injection is not a replacement for fine grained object lifetime management. The design thus reflects real world requirements for wiring enterprise C++ applications.

Design

Meta-types

The table below lists the previous meta-types again, along with comments raised during the design phase with regard to binding creation.

Meta-type	Provide binding?	Comments
T	Yes	Copy elision / copy semantics of stack based new instances.
unique_ptr<T>	Yes	Unique ownership of heap based polymorphic new instances.
T &	Yes	Warn in the documentation that object lifetime is the responsibility of the end developer.
const T &	Yes	Warn in the documentation that object lifetime is the responsibility of the end developer.
shared_ptr<T>	Yes	Shared ownership of heap based polymorphic singletons and thread-local singletons.
shared_ptr<const T>	Yes	Const singletons could be useful and their inclusion does not impact the design.
*T **	No	Raw pointers should be managed inside pointer containers.
*const T **	No	Raw pointers should be managed inside pointer containers.
*T * const*	No	Raw pointers should be managed inside pointer containers.
*const T * const*	No	Raw pointers should be managed inside pointer containers.
const T	Promote	The semantics are identical to non-const new value instance provision.
const unique_ptr<T>	Promote	The semantics are identical to non-const new polymorphic instance provision.
unique_ptr<const T>	Promote	The semantics are identical to non-const new polymorphic instance provision.
const unique_ptr<const T>	Promote	The semantics are identical to non-const new polymorphic instance provision.
const shared_ptr<T>	Promote	The semantics are identical to non-const shared pointer to T.
const shared_ptr<const T>	Promote	The semantics are identical to non-const shared pointer to const T.

The use of *T*, *std::unique_ptr<T>*, and *std::shared_ptr<T>* meta-types for non-polymorphic values, polymorphic instances, and singletons respectively was natural from the outset.

A decision that was taken during the design of the injector was to raise the severity of using raw pointer bindings to a compile time error, via static assertions. Thus any bindings defined with raw pointers result in a compile time error, along with an error message that proposes using *Unique* or *Shared* bindings instead.

One consideration was whether reference bindings should be allowed, or whether the lifetime management of this would open up the risk of dangling references. The conclusion was that references should be provided, but the implications of providing references from the injector should be clearly discussed in the documentation.

Another consideration was how to implement const versions of long lived objects (i.e. *const BaseT &* and *std::shared_ptr<const BaseT>*, including provision for promoting a non-const binding to a const binding when no suitable const binding has been registered.

The final design thus provides the following types of non-const object.

Type of object	Comments
non-polymorphic instances *T*	Non-polymorphic instances are stack based values produced from default construction, prototype copying, and provider bindings. A new instance is created on each call.
polymorphic instances *std::unique_ptr<T>*	Polymorphic instances are heap based abstract values. A new instance is created on each call.
polymorphic references *T &*	Long lived provided objects, managed by the application. The injector plays no part in lifetime management, and assumes that the referenced object specified in the configuration will live longer than the injector and the consumers of references supplied by the injector.
polymorphic thread-local singletons *std::shared_ptr<T>*	Thread-local singletons are, amongst other things, useful for tunnelling information through a call stack without the need for explicit and repeated call parameters or concurrent techniques.
polymorphic singletons *std::shared_ptr<T>*	Singletons form the basic wiring of the software application. Lazy singletons (the default) allow optional singletons to be defined in configuration but only instantiated if requested.

In addition to non-const bindings, the injector provides the following types of const object.

Type of object	Comments
polymorphic references *const T &*	Const version of the reference binding.
polymorphic thread-local singletons *std::shared_ptr<const T>*	Const version of the thread-local singleton binding.
polymorphic singletons *std::shared_ptr<const T>*	Const version of the singleton binding.

Const promotions

Whilst certain const meta-type forms are not included in the previous list, the injector does nevertheless implement const promotion. When a binding request for a const type is not available, a suitable non-const type will be provided instead if available. This applies equally to the non-polymorphic and polymorphic new instance binding types which do not support const bindings in the configuration.

The following table details these non-const to const binding promotions.

Requested meta-type	Default provided meta-type	Promoted provided meta-type
const T	-	*T*
const std::unique_ptr<T>	-	*std::unique_ptr<T>*
std::unique_ptr<const T>	-	*std::unique_ptr<T>*
const std::unique_ptr<const T>	-	*std::unique_ptr<T>*
const T &	*const T &*	*T &*
const std::shared_ptr<T>	-	*std::shared_ptr<T>*
std::shared_ptr<const T>	*std::shared_ptr<const T>*	*std::shared_ptr<T>*
const std::shared_ptr<const T>	*std::shared_ptr<const T>*	*std::shared_ptr<T>*

Performance

Once an application is compiled with optimisation, each get-instance call in an injector collapses down to a lookup in the binding map and a virtual method call on the looked up binding object. The keys used in the binding map contain a type index encapsulating the meta-type and const qualifier, plus a UTF-8 string for the name. The hash thus consists of the type index hash combined with the hash of the string.

The injector's binding map is only mutated during the configuration phase of the injector instantiation. Each injector's binding map is thus *const* and consequently an unsynchronised hash map is used internally. The injector is thread safe and no synchronisation is used in get-instance calls.

Planned C++20 features

One feature that has not been implemented in the current version of the Balau injector is compile time binding keys. The idea of building the binding keys at compile time via the get-instance typename and string literal is not achievable in a simple way in C++17.

In a get-instance call, the type argument encapsulates all the information required for the key apart from the name, i.e. binding meta-type, const qualifier, and typeid. In order to provide compile time binding keys, the compile time name would also need to be specified as a template argument.

```
// This is not possible in C++17.
auto obj = injector->getInstance<int, "blah">();
```

A fully compile time key would also allow hashes to be precalculated, reducing the binding lookup to a modulus calculation and one or more equals calls on the hash map bin contents.

Whilst there are fudges and hacks to get the above kind of working in C++17, it was decided that runtime binding keys would be sufficient until C++20 is released. C++20 should allow string literal template arguments to be used, allowing the above code to be used without any hacks.

The current runtime binding key get-instance methods will remain as they are. The addition of compile time binding key get-instance methods will be implemented within an *#ifdef* block, allowing continued use of the library with a C++17 compiler.

Environment configuration

Overview

Introduction

The *EnvironmentConfiguration* class provides injectable environment configuration from programming language agnostic hierarchical property files.

Typically, a server application will be run as one or more separate processes spread across a group of machines. Each running process will be configured according to a specified environment. In order to configure the application differently for each environment, the *EnvironmentConfiguration* class provides a convenient way of binding externally sourced hierarchical environment properties into the injector, ready for injection into dependent classes as typed named values and named composite *EnvironmentProperties* instances.

The *EnvironmentConfiguration* class works with hierarchical property files (see the property parser chapter for more information on defining hierarchical property files). A URI referencing a properties file is specified as a constructor argument of the *EnvironmentConfiguration* class or an implementing class of the *EnvironmentConfiguration* class. The referenced properties file is parsed and appropriate bindings are created when the environment configuration instance is called by the instantiating injector.

In addition to string value properties, the environment configuration framework supports typed configuration properties via type specifications. The use of type specification files allows a single set of hierarchical environment configuration type specification files to be used across multiple applications written in multiple programming languages.

Type specifications may also have default values attached to them. This allows sensible defaults that apply to all environments to be specified in a single location, preventing the need for complex environment configuration property files and consequential logical coupling across multiple environments and applications.

Type specification files and property value files are conceptually similar to classes and instances. A type specification file provides a hierarchical typed contract. A property value file provides hierarchical instance values that fulfill a type specification contract. This analogy is not exact, as type specification files can provide default values for instantiation and property value files can provide values that default to string types when no matching type specification is provided. The hierarchical configuration design thus provides a looser contract than the class-instance contract.

The Balau library provides a C++ implementation of the environment configuration and environment properties support classes. Java based environment configuration classes are also planned, to support *Guice* and *Spring* based applications.

Usage patterns

There are three ways to use the environment configuration class:

- derive from the *EnvironmentConfiguration* class and hard wire hierarchical property type specifications inside the configure() method of the derived class;
- instantiate an *EnvironmentConfiguration* instance directly, by specifying one or more hierarchical property type specification sources in addition to the input environment properties sources.
- derive from the *EnvironmentConfiguration* class, hard wire hierarchical property type specification declarations inside the configure() method of the derived class, and specify one or more hierarchical property type specification sources in addition to the input environment properties source.

The first approach places the hierarchical property type specifications directly within the C++ code. An advantage of this approach is that any type that has a corresponding *fromString* function can be used for value property types parsed by the environment configuration, without the need for explicit registration before injector creation.

The second approach places the hierarchical type specifications within type specification definition files. The advantage of the second approach is that the environment configuration type specifications are defined within an IDL (which is itself also in the hierarchical property format). Environment configuration type specification files may thus be defined once and used for multiple software applications written in multiple languages, without needing to redefine the type specifications or the environment configuration files.

The disadvantage of the second approach is that all custom types (i.e. types not pre-registered in the Balau library) referenced in the type specification files must be registered with the consuming C++ applications before creating the injector. This can be achieved via the following function calls.

- *EnvironmentConfiguration::registerValueType<T>*
- *EnvironmentConfiguration::registerUniqueType<T>*

Such registration of custom property types will not be necessary in other languages such as Java, where reflection can be used to resolve string to type mappings.

The third approach mixes the first two approaches together. This involves deriving from the *EnvironmentConfiguration* class and also passing one or more type specification properties files to the base class constructor, in addition to implementing the *configure* method.

With the first and third approaches, additional validation logic can be placed in the *configure* method if required. This places validation logic specific to specific environment configuration inside the same class that generates the injector bindings for it.

With all three approaches, type specifications are optional. If a value property is present in a specified property file and there is no corresponding type specification, a *std::string* property will be created (*std::string* being the default property value type).

Quick start

#include <Balau/Application/EnvironmentConfiguration.hpp>

Properties

An example hierarchical property file used for environment configuration looks similar to the following.

```
http.server.worker.count = 8

file.serve {
    location      = /
    document.root = file:src/doc
    cache.ttl     = 3600
}
```

The above is an example from the Balau HTTP server tests.

Hard wired specifications

Hard wiring the environment configuration type specifications is best limited to applications that either:

- do not share their environment configuration with other applications; or
- share their environment configuration with other C++ applications via a shared C++ library.

The creation of an environment configuration class with hard wired type specifications consists of deriving from the *EnvironmentConfiguration* base class, and implementing the *configure* method. Environment configuration type specifications that correspond to the properties in the referenced property file are placed within the *configure* method.

The following is an example environment configuration class implementation for the above example property file.

```
// Environment configuration for the example HTTP file server.
class EnvConfig : public EnvironmentConfiguration {
    public: EnvConfig(const Resource::Uri & input) : EnvironmentConfigurat

    public: void configure() const override {
        value<int>("http.server.worker.count");

        group("file.serve"
            , value<std::string>("location")
            , unique<Resource::Uri>("document.root")
            , value<int>("cache.ttl")
        );
    }
};
```

IDL based specifications

IDL based environment configuration type specifications are best used for applications that:

- are written in multiple programming languages (and thus require language agnostic type specifications);
- are maintained by multiple teams each with a different release schedule.

The IDL based method for specifying environment configuration involves the external creation of environment configuration type specification files, referenced via URI.

The creation of a type specification file consists of creating a hierarchical property file that is similar to the environment configuration property file. Instead of specifying property data values, the type annotations are specified.

Direct instantiation of the *EnvironmentConfiguration* class involves passing one or more environment configuration type specification files, referenced via URI to the constructor of the *EnvironmentConfiguration* class. The type specification files are cascaded together, and the resulting hierarchical type specifications are used as if they were specified within the *configure* method.

The following properties file is an example environment configuration type specification for the previous properties data file.

```
http.server.worker.count : int

file.serve {
    location      : string
    document.root : uri
    cache.ttl     : int
}
```

In this example, the *:* separator has been used in order to accentuate that the property values are type annotations. Alternative = or whitespace separators can also be used if preferred.

Property type specification files look very similar to property data files, as they have a similar hierarchical structure.

In order to use the type specification file, the *EnvironmentConfiguration* class is instantiated, specifying the type specification file and the property data file to the constructor.

```
// Direct instantiation of the EnvironmentConfiguration class.

auto envProps = Resource::File("path/to/env/env1.hconf");

auto specs1 = Resource::File("path/to/type/specs1.thconf");
auto specs2 = Resource::File("path/to/type/specs2.thconf");

auto envConf = EnvironmentConfiguration(envProps, specs1, specs2);
```

The *EnvironmentConfiguration* class' constructor is variadic, thus multiple type specification URIs may be specified.

Mixed specifications

Hard wired and IDL based environment configuration type specifications can be mixed in an environment configuration derived class.

If a mixed type specification is used in an environment configuration, it is possible that duplicate type specifications are provided for a property (one in the hard wired configuration and another in the type specification file). When this occurs, the hard wired type specification / default value takes precedence over the file based one. It is thus possible to override a file base type specification by adding a hard wired type specification with an identical name and hierarchy position in the derived class configuration method.

Default values

Both hard wired and IDL environment configuration type specifications may have default values attached to them. This allows concise environment configuration data files to be created, by specifying only the differences between the defaults and the required values for that environment.

The following example is a copy of the previous hard wired example environment configuration class, with default values attached to the properties that can have sensible defaults.

```
// Example environment configuration with sensible defaults.
class EnvConfig : public EnvironmentConfiguration {
    public: EnvConfig(const Resource::Uri & input) : EnvironmentConfigurat

    public: void configure() const override {
        value<int>("http.server.worker.count", 8);

        group("file.serve"
            , value<std::string>("location")
            , unique<Resource::Uri>("document.root")
            , value<int>("cache.ttl", 3600)
        );
    }
};
```

The following example is a copy of the previous IDL example environment configuration type specifications file, with default values attached to the properties that can have sensible defaults.

```
http.server.worker.count : int = 8

file.serve {
    location      : string
    document.root : uri
    cache.ttl     : int = 3600
}
```

Application creation

Injector

In order to create an injector with both application and environment configuration, the environment configuration instance(s) are passed to the injector's create function in the same way as is done with the application configuration instance(s).

```
// Create an injector from a single application configuration
// and a single environment configuration.
auto injector = Injector::create(AppConfig(), EnvConfig(envProps));
```

The above code will parse the contents of the properties file and will build a corresponding set of bindings in the injector that correspond to the root properties (simple and composite) within. Simple properties become named value or unique bindings of the type specified in the environment configuration declaration, or *std::string* if no type specification was made for that property. Composite properties become named shared bindings of type *EnvironmentProperties*.

Any issues encountered during the build will be flagged via an *EnvironmentPropertiesException*.

All properties contained within a composite property become bindings within the resulting *EnvironmentProperties* instance. This may include other composite properties that themselves may contain their own bindings. *EnvironmentProperties* instances have a similar public API to part of the injector's API. Three get-instance calls are available:

Get-instance call	Instances obtained
getValue	Value property non-polymorphic values
getUnique	Value property polymorphic values
getComposite	Composite properties

Unlike the bindings of the injector class (which may include bindings of type *EnvironmentProperties*), the bindings contained within *EnvironmentProperties* instances are not automatically injected into other dependencies. Due to this, the *EnvironmentProperties* class does not have *getInstance* methods (which are used in the automatic injection functions of injectable classes).

The string to object conversion mechanism used for value property non-polymorphic and polymorphic values is Balau's universal *from-string* function. More information on the universal *from-string* function is available in the characters and strings chapter. For non-polymorphic value types, a from-string function should be defined with a reference to the destination object. For polymorphic value types, a from-string function should be defined with a reference to a *std::shared_ptr<T>* destination object.

Once the environment property bindings have been created, the root set of simple and composite properties can be used in the same way as all other bindings in the injector, including automatic injection into other dependencies.

Application

In order to use an environment's properties file(s), the C++ application's main function must provide a path to the environment's home directory / property file location(s). This can be achieved by various methods, a couple of simple ones being:

- passing the environment's home directory via a command line argument;
- passing an environment identifier via a command line argument and resolving the environment's home directory via a simple mapping.

The *EnvironmentConfiguration* class is designed to fail immediately if an environment configuration property file is not well formed, thereby preventing the application from starting up with an invalid configuration.

Credentials

Part of the environment configuration supplied to an application will be one or more credential properties. Unlike other environment configuration properties, credentials properties should not be checked into VCS and will thus exist in one or more separate properties files. This approach allows credentials information to be private to system administrators.

Applications that require credentials information should thus have at least two environment configuration properties files:

- one for the main environment configuration (checked into VCS);
- a second one containing credentials information and with a similar hierarchical structure to the main configuration file.

During creation of the environment configuration injector bindings, the main environment configuration and the credentials environment configuration will be merged together to form a single set of hierarchical properties that will be transformed into bindings.

Property type IDL

Property type specifications are defined in the hierarchical property format. In the IDL, the physical structure of the type specification hierarchy is similar to the hierarchy of corresponding environment configuration data files.

The following type strings are currently pre-registered for the environment configuration framework:

- byte;
- short;
- int;
- long;
- float;
- double;
- string;
- char;
- boolean;
- uri.

All the pre-registered types apart from the *uri* type are non-polymorphic value types (values obtained via subsequent *getValue* calls). The *uri* type is a polymorphic *unique* value type (*std::unique_ptr* values obtained via subsequent *getUnique* calls).

The environment configuration types map to the following C++ types.

type string	C++ type
byte	signed char
short	short
int	int
long	long long
float	float
double	double
string	std::string
char	char
boolean	bool
uri	std::unique_ptr<Resource::Uri>

As environment configuration type specifications have been designed to be programming language independent, unsigned integer types are not included as pre-registered types. If unsigned integer type specifications are required for a Balau based C++ application, they can be manually registered by calling the *EnvironmentConfiguration::registerUnsignedTypes* function. Calling this function before creating the injector will register the following types:

type string	C++ type
unsigned byte	unsigned char
unsigned short	unsigned short
unsigned int	unsigned int
unsigned long	unsigned long long

Other non-standard types may also be manually registered by calling one of the following functions for each custom type, before creating the injector.

- EnvironmentConfiguration::registerValueType<T>
- EnvironmentConfiguration::registerUniqueType<T>

The first function should be used to register non-polymorphic value types (for values obtained via subsequent *getValue* calls). The second function should be used to register polymorphic *unique* value types (for *std::unique_ptr* values obtained via subsequent *getUnique* calls).

In addition to the type string used in type specification files, the polymorphic function takes a cloner function. This function will be used to clone the *std::unique_ptr* prototype value on each subsequent call to *getUnique*.

The full signatures of the two functions are as follows.

```
///
/// Add a non-polymorphic type custom property binding builder
/// factory to the global property binding builder factory map.
///
template <typename ValueT>
void registerValueType(const std::string & typeString);

template <typename BaseT>
using UniquePropertyCloner = std::function<
    std::unique_ptr<BaseT> (const std::unique_ptr<const BaseT> &)
>;

///
/// Add a polymorphic type custom property binding builder
/// factory to the global property binding builder factory map.
///
template <typename BaseT>
void registerUniqueType(const std::string & typeString,
                        const UniquePropertyCloner<BaseT> & cloner);
```

An example of a hierarchical property type specification file follows.

```
http.server.worker.count : int

file.serve {
    location      : string
    document.root : uri
    cache.ttl     : int
}
```

In the above example, the *:* separator has been used in order to accentuate that the property values are type annotations. Alternative = or whitespace separators can also be used if preferred, although the visual representation of default values will look less attractive if an = token is used for both the property delimiter and the devault value.

Property type specification files look very similar to property files, as they have a similar hierarchical structure.

As discussed previously, if a type specification is absent for a particular property, a binding for the property will be created with type *std::string*. Type specification files may thus only include type specifications for properties that have types other than the default *std::string*.

Regardless of type, it can be useful to include type specifications for certain *std::string* properties, in order to specify default values for those properties. The following type specification file is the same as the previous one, but with default values for worker count and cache TTL.

```
http.server.worker.count : int = 8

file.serve {
    location      : string
    document.root : uri
    cache.ttl     : int = 3600
}
```

Configuration cascading

An application that uses the environment configuration framework will consume the following information:

1. file based type specifications and default values;
2. class based type specifications and default values;
3. file based property values.

The information from these three sources is merged together to form a hierarchical set of *value*, *unique*, and *shared* bindings in the injector.

There are thus two merges that occur during the configuration of the injector:

- merging of type specification / default value hierarchies;
- merging of default values from the previous merge and property values.

The first type of cascade consists of the priority merging of type information and associated default values, sourced from potentially multiple configuration producing applications. Type specification / default value overrides can thus be specified, either in additional type specification files or hard coded in an *EnvironmentConfiguration* derived configuration class.

The second type of cascade consists of the merging of the environment's property values with the default values resulting from the type specification priority merge. Unlike the restrictions on the main property values, this value merge allows duplicate values to be specified. The default values in the merged type specifications are overridden by the environment's property values.

Example configuration

A full example is presented here (taken from the Balau tests). The example has a single type specification file, a single derived environment configuration class with hard wired type specifications, and a property value file.

The specification file contents is as follows.

```
http.server.worker.count : int = 6
value.multiplier         : double = 123.456

file.serve {
    location      : string
    document.root : uri
    cache.ttl     : int = 10000

    options {
        identity : string = Balau Server
        404      : uri    = file:404.html
    }
}
```

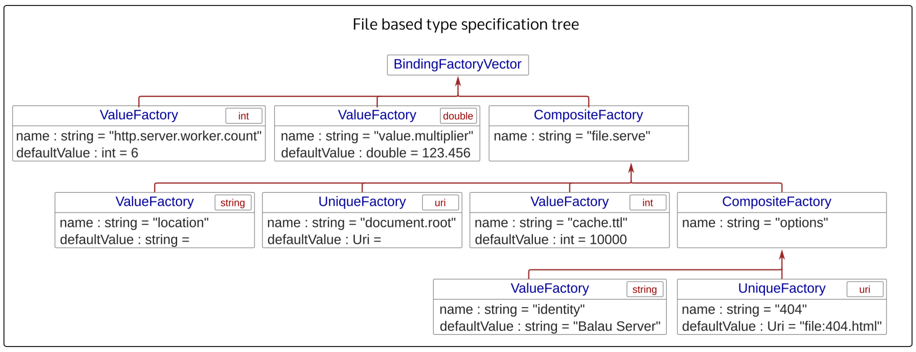
The hard wired type specification class is as follows.

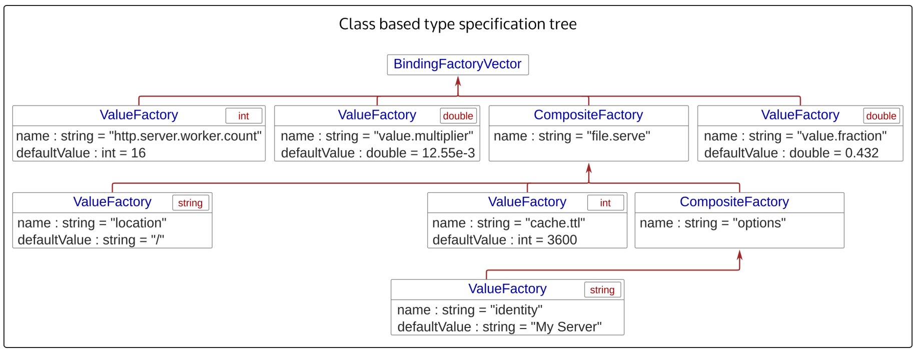
```
struct EnvConfig : public EnvironmentConfiguration {
    EnvConfig(const File & env, const File & spec)
        : EnvironmentConfiguration(env, spec) {}

    void configure() const override {
        value<int>("http.server.worker.count", 16);
        value<double>("value.multiplier", 12.55e-3);

        group("file.serve"
            , value<std::string>("location", "/")
            , value<int>("cache.ttl", 3600)
            , group("options"
                , value<std::string>("identity", "My Server")
            );
        );

        value<double>("value.fraction", 0.432);
    }
};
```

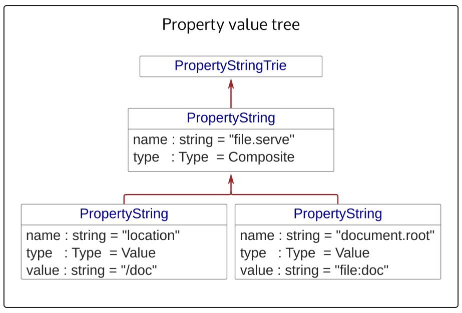
The environment's property value file is as follows.

```
file.serve {
    location      = /doc
    document.root = file:doc
}
```

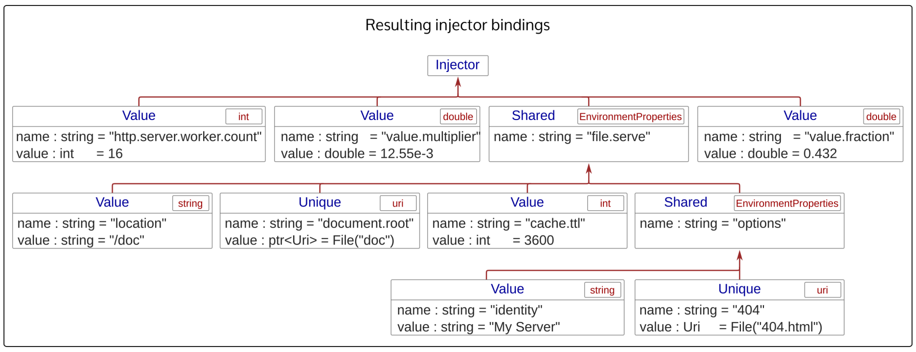
The intermediate trees generated from the three sources are illustrated in the following diagrams.


File based type specification tree
BindingFactoryVector
ValueFactory int
name : string = "http.server.worker.count"
defaultValue : int = 6
ValueFactory double
name : string = "value.multiplier"
defaultValue : double = 123.456
CompositeFactory
name : string = "file.serve"
ValueFactory string
name : string = "location"
defaultValue : string =
UniqueFactory uri
name : string = "document.root"
defaultValue : Uri =
ValueFactory int
name : string = "cache.ttl"
defaultValue : int = 10000
CompositeFactory
name : string = "options"
ValueFactory string
name : string = "identity"
defaultValue : string = "Balau Server"
UniqueFactory uri
name : string = "404"
defaultValue : Uri = "file:404.html"


Class based type specification tree
BindingFactoryVector
ValueFactory int
name : string = "http.server.worker.count"
defaultValue : int = 16
ValueFactory double
name : string = "value.multiplier"
defaultValue : double = 12.55e-3
CompositeFactory
name : string = "file.serve"
ValueFactory double
name : string = "value.fraction"
defaultValue : double = 0.432
ValueFactory string
name : string = "location"
defaultValue : string = "/"
ValueFactory int
name : string = "cache.ttl"
defaultValue : int = 3600
CompositeFactory
name : string = "options"
ValueFactory string
name : string = "identity"
defaultValue : string = "My Server"



Once merged, the resulting hierarchical bindings defined in the injector are illustrated in the following diagram.


Resulting injector bindings
Injector
Value int
name : string = "http.server.worker.count"
value : int = 16
Value double
name : string = "value.multiplier"
value : double = 12.55e-3
Shared EnvironmentProperties
name : string = "file.serve"
Value double
name : string = "value.fraction"
value : double = 0.432
Value string
name : string = "location"
value : string = "/doc"
Unique uri
name : string = "document.root"
value : ptr<Uri> = File("doc")
Value int
name : string = "cache.ttl"
value : int = 3600
Shared EnvironmentProperties
name : string = "options"
Value string
name : string = "identity"
value : string = "My Server"
Unique uri
name : string = "404"
value : Uri = File("404.html")

Design

This section provides a summary of some aspects of the philosophy and design of the hierarchical environment configuration framework. It is not necessary to read this section in order to use the framework.

Overview

The aim of the Balau environment configuration framework is to provide a way to specify environment specific, injectable configuration that:

- is type safe, via inter-environment type specifications;
- provides the possibility of specifying sensible, inter-environment defaults for properties;

- allows hierarchical property data to be specified;
- provides a hierarchical format that facilitates the specification of independent, composite hierarchical property subsets;
- has a familiar syntax;
- is programming language independent.

Background

Environment configuration in enterprise applications has been achieved in a variety of ways. One common technique is to supply environment configuration as a flat file containing key-value pairs. These key-value pairs are then loaded into the application injector to form injectable named string values. One common format for this approach is the *.properties* format often used in Java applications.

Other bespoke approaches to environment configuration include hierarchical configuration formats. Examples of these include XML based configuration (such as the format used in the configuration files of the Apache HTTP server), and curly bracket based hierarchical configuration blocks (the configuration files of the Nginx server being an example).

More recently, an elaboration on the traditional key-value approach has been to use YAML as a more visually informative format for key-value properties. This approach also allows string representations of complex values to be represented within the same physical structure as the key-value property structure (i.e. value lists defined in the standard YAML format).

Requirements

The key requirements determined during the design phase of the environment configuration framework were:

1. provide a mechanism for composite properties, which represent standalone environment configuration property subsets, injectable as a single *EnvironmentProperties* class instance;
2. allow types to be optionally specified for key-value properties;
3. provide a mechanism for specifying sensible property defaults;
4. separate configuration data into inter-environment data (property types, default property values) and intra-environment data (environment specific property values);
5. provide an *include* mechanism, which allows a hierarchical property file to be split across multiple physical files;

6. do not mix hierarchical structure with string representations of typed values (i.e. string representations of a complex data structure assigned to a value property);
7. do not implement required/optional specifications for properties;
8. base the hierarchical property format on a well known non-hierarchical property format, with minimal differences to the original format;
9. do not infer the hierarchical structure from indentation, use the well known "{" and "}" block delimiter token pair instead;
10. use the standard string type of the implementing programming language (e.g. *std::string* for C++) for value properties that do not have a corresponding type specification.

File format

The resulting format chosen for both hierarchical environment configuration data files and type specification files is described in detail in the property parser chapter. The format is based upon a hierarchical extension to the *.properties* file format. Composite properties are defined via "{" and "}" delimited blocks. The *include* mechanism uses the "@" token to specify a URI to be included.

Other than the addition of the special "{", "}" and "@" characters and the corresponding non-special, escaped "\{", "\}" and "\@" character pairs, the hierarchical property format is identical to the original non-hierarchical property format.

Specification files

Environment configuration type specification files represent inter-environment configuration data. This data is defined by an owning application and is shared across multiple environments and consuming applications. The two pieces of information provided for each value property are:

- the property's value type;
- the property's optional default value.

Value files

Environment configuration value files represent intra-environment configuration data. This data is unique to an environment, but can be shared across multiple applications for that environment. Environment configuration data files contain string representations of each value property contained in the configuration hierarchy.

If a type specification for a value property is specified in the cascade of the supplied type specification files, the value property will be typed. Otherwise, the property will be bound as a string.

Configuration cascading

An important part of the environment configuration design was getting the semantics of merging multiple type specification / default value and property value sources together. This involves the merging of information of the following types:

1. file based type specifications and default values;
2. class based type specifications and default values;
3. file based property values.

There are three consequential information merges that could possibly occur:

- merging type specification / default values;
- merging property values;
- merging default values and property values;

Finalising the exact semantics of each type of information cascade was a major part of the design process of the environment configuration framework.

The first type of cascade consists of the priority merging of type information and associated default values, sourced from potentially multiple configuration producing applications. During the design analysis phase, it was concluded that duplicate type specification / default values should be allowed. This allows type specification / default value overrides to be specified, either in additional type specification files or hard coded in an *EnvironmentConfiguration* derived configuration class.

The second type of cascade would consist of the priority merging of the property values of the environment from multiple property value sources. This type of merging is not supported in the environment configuration framework. Taking into account that each property value becomes an injector binding, the presence of duplicate property values is effectively the same as the presence of duplicate *value* or *unique* application configuration bindings. As the injector does not allow such duplicate bindings, this restriction is carried through to property values. This is the case both within a single property value file processed by an *EnvironmentConfiguration* instance (property value duplication), and across multiple application and environment configuration instances (binding duplication) potentially specified to the injector.

The third type of cascade consists of the merging of the environment's property values with the default values resulting from the type specification priority merge. Unlike the restrictions on the main property values, this value merge does allow duplicate values to be specified. The default values in the merged type specifications are overridden by the environment's property values.

Consequently, there are only two types of information merge that occur in the environment configuration framework:

- merging type specification / default values;
- merging default values and property values;

No required properties

One aspect of the design that matured during the development of the environment configuration property framework was whether type specification files should have provision for required/optional property annotations.

Overview

When considering this proposal, the first impressions of all involved developers were that the capability of specifying which environment properties are required would improve the correctness and safety of consuming applications. This however changed after a more deep consideration of the implications of such a feature, including when taking into account the releases cycles of multiple applications in an enterprise. It was subsequently concluded that required/optional annotations would have a net negative effect on the development and release process.

When a single version of a consuming application is made in isolation, required/optional property annotations are clearly an advantage. Such annotations would provide both a visual indication for developers and a validation mechanism at application startup.

However, when a more global consideration is made which takes into account multiple environments, multiple applications, and multiple application versions, the required/optional annotation feature forms a likeness to the abandoned required/optional designs of wire data formats such as Protocol Buffers and Apache Thrift.

Analysis

The stated advantages of required/optional annotations in the environment configuration framework were:

- a validation mechanism run during application startup;

- a clear visual indication for developers that are creating or modifying environment configurations.

The primary disadvantage of required/optional annotations in the environment configuration framework is that once a property is marked as required in a producing application's type specification declarations, the release cycles of the suite of consuming applications in an enterprise would be constrained to be synchronised. Such a release cycle synchronisation is extremely inefficient in an enterprise of any complexity, and in the worst case would result in breakages in the contracts between applications.

This disadvantage is exactly the same disadvantage which led to the abandonment of required/optional annotations in wire data formats such as Protocol Buffers and Apache Thrift .

Alternatives

When examining in more detail the overall design, the stated advantages of required /optional annotations can be achieved via other means.

Validation

Modern enterprise quality software development is performed in the context of test driven development. When using a dependency injection framework, development should include testing of the application configuration. Such configuration testing is described in the injector chapter for the Balau injector.

When the injector configuration is tested, the validation mechanism provided via required /optional annotations is rendered unnecessary.

If, in exceptional circumstances, additional runtime validation is required, this additional validation can be placed within the *configure* method(s) of the *EnvironmentConfiguration* derived classes of the application.

Visual indications

Whilst visual indications of required properties for developers would be useful, the lack of required/optional annotations in type specification files can be mostly mitigated by prioritising the use of sensible property value defaults. The design of the hierarchical environment configuration framework includes provision for inter-environment property defaults, specified within the type specification files of producing applications.

If use of the default values feature is made a primary part of the development process, new properties that are introduced into the release of a component will not require the developers of consuming applications to "mend" their applications when upgrading to new versions of the consumed type specification files. Instead, the default values of new properties will be picked up automatically. Subsequent overriding of the new properties' default values can then be made if necessary at a later date.

Logger

Overview

A logging framework, with configurable loggers via logging namespaces. Loggers output to C++ streams specified via the logging namespace configuration. Logging namespaces inherit their parent configuration via a cascading inheritance.

Logging namespaces are configured via standard environment configuration syntax. This can be supplied statically (via a logging configuration file), or dynamically by calling the logging system *configure* method.

Quick start

#include <Balau/Logging/Logger.hpp>

Environment configuration: logging

Logging messages

Logging to Balau loggers is similar to that of other logging systems.

```
// A debug message.
log.debug("An object: {}", obj);

// An info message.
log.info("Hello, world!");

// An error message.
log.error("Something went wrong. Data: {} / {}", data1, data2);
```

The first argument is a *const char ** or *std::string_view* string that specifies the message. The *{}* placeholders in the message are replaced by the string conversions of the remaining arguments.

If source code file and line number information is required, the logging macros should be used instead. The definitions of these macros contain the standard *__FILE__* and *__LINE__* macros that provide file and line number information.

```
// A debug message with file and line number information.
BalauLogDebug(log, "An object: {}", obj);

// An info message with file and line number information.
BalauLogInfo(log, "Hello, world!");

// An error message with file and line number information.
BalauLogError(log, "Something went wrong. Data: {} / {}", data1, data2);
```

If you do not wish to see the prefix *Balau* on each of your logging statements, new defines can be created which alias these macros.

There are six logging levels: *trace*, *debug*, *info*, *warn*, *error*, and *none*.

Whether a log message is output to the logging stream depends on whether the logging namespace is enabled for that logging level. This depends on the logging configuration.

The *{}* placeholders are replaced with the result of the toString call of the object(s) supplied in the logging call. It is thus necessary that suitable *toString* functions are defined for all object types passed to logging calls.

In order for the compiler to pick up the correct *toString* functions during the function template instantiation, the header file(s) containing the function(s) must be included before the *Logger.hpp* header is included.

In addition to the standard logging methods that accept a variable number of parameters, there is a function based logging method for each logging level. These methods accept a function that is used to generate the message to log.

```
// An info message via a lambda.
log.info([&v1, &v2] () { return toString("Value = ", v1 * v2); });
```

The function based logging methods are useful when derived arguments need to be logged. If a function based logging method were not used, the code used to derive the arguments would execute, regardless of whether the message is needed or not, as prescribed by the logging level of the logger.

In order to use loggers, there are two tasks required:

- configure the logging system namespaces (optional);
- obtaining references to loggers.

These tasks are discussed in reverse order below.

Logger references

Loggers are identified via logging namespaces. A logging namespace is a string of dot delimited identifiers. Typically, the application name or the reverse domain name of the company is used as the logging namespace prefix, i.e. *"balau"* or *"com.borasoftware"*. Namespaces are case sensitive.

Loggers are owned by the logging system and can be obtained by reference, by calling the *Logger::getLogger* function.

```
// Get a logger for the "balau.network" logging namespace.
Logger & log = Logger::getLogger("balau.network");
```

Logger references are typically set up as static fields of classes, but instance references and local variables are also possible if the use case requires it.

```
////////// In header file. //////////

class A {
    // Static member logger reference declaration.
    private: static Balau::Logger & log;
};

/////////// In body file. ///////////

// Static member logger reference definition.
Balau::Logger & A::log = Balau::Logger::getLogger("balau.network");
```

Logging configuration

The configuration of loggers is determined by the configuration of the logging namespaces. Logging namespaces inherit their parent configuration via a cascading priority inheritance.

By default, the logging system configures itself automatically to log info and warn level messages to stdout, and error level messages to stderr.

The default logging message format is:

```
%Y-%m-%d %H:%M:%S [%thread] %LEVEL - %namespace - %message
```

Details of the message format placeholders are provided in the detailed documentation.

There are two ways of providing a custom logging configuration:

- create a *balau-logging.hconf* configuration file in the same folder as the application's executable;

- call Balau::Logger::configure(std::string) within the application.

If a logging configuration file is provided, the contained configuration will be used from the start of the application's execution. If logging configuration is provided via a *Balau::Logger::configure* call, the logging system will use the default configuration until the call is made.

The logging configuration is standard environment configuration. Composite property names specify the logging namespaces. The value properties defined within a composite property provide the logging configuration for that namespace.

Usage

Configuration

There are two ways to supply the logging configuration.

The implicit way to configure the logging system is to provide a *balau-logging.hcon* configuration file in the application binary directory. This will be automatically picked up by the logging system.

The explicit way to configure the logging system is to provide logging configuration via a call to:

```
Balau::Logger::configure(const std::string & conf);
```

The string argument passed in the call contains the logging configuration. This logging configuration can be read from a file in a custom location, or may be generated by the application as deemed appropriate by the caller.

The logging system can be reconfigured via subsequent calls to the above function. Calling the following function will lock the configuration for the lifetime of the application execution:

```
Balau::Logger::lockConfiguration(bool throwOnReconfigure = false);
```

Subsequent calls to the configure function will then either be ignored (default behaviour) or will throw a LoggingConfigurationException if throwOnReconfigure is set to true.

Logging reconfiguration provides eventual consistency. Logging during reconfiguring will result in stable but non-deterministically configured logging. The logging format of a particular logger during reconfiguration will either be the previous configuration or the new configuration, but not a mixture of the two. Similarly, the stream(s) written to by a logger during reconfiguration may be the previously specified ones or the newly specified ones.

Logging configuration calls can be made in static contexts, including within static initialisation blocks that are each executed in a non-deterministic order.

If no *balau-logging.hconf* file is found and no call to configure is made, the logging system will log INFO and WARN level messages to stdout and ERROR messages to stderr.

If the parsed logging configuration from *balau-logging.hconf* is invalid, an error message will be written to stderr and loggers associated with the affected namespaces will be configured to write to stdout for all invalidly configured logging levels. If the format specification for a namespace contains an invalid format specifier, an error message will be written to stderr and the the invalid specifier will be output verbatim in subsequent log messages to affected loggers.

Logger instances

Loggers are obtained via the static getLogger call in the logger class:

```
Balau::Logger & logger = Balau::Logger::getLogger(const std::string & logg
```

The parameter 'loggingNamespace' is the logging namespace that determines the configuration of the logger (logging level, streams, message format).

The *Logger* class also contains a static function *globalLogger()*. This logger references the logger associated with the global namespace. This logger may be useful when logging to the global namespace is all that is required.

Calls to getLogger() can be in static or instance contexts. Normally such calls are placed in static contexts.

```
//////////// Header file ////////////

class A {
    // The class' logger.
    private: static Balau::Logger & logger;

    // main class declaration..
};

///////////// Body file /////////////

// Get the logger associated with the logging namespace "A".
Balau::Logger & A::logger = Balau::Logger::getLogger("A");
```

Loggers should always be maintained as references. Loggers are not owned by the caller, and must not be deleted or placed within pointer containers.

The Balau logging system will not throw any exceptions from the *getLogger* call, other than if there is a fatal error such as an out of memory issue.

Startup and shutdown

The logging system is first configured when the first call to *Logger::getInstance* or *Logger::configure* is made. Subsequent calls to *Logger::configure* will reconfigure the logging system.

The logging system state is maintained in a lazy singleton that is instantiated on the first call to *Logger::getInstance* or *Logger::configure*. It is thus safe to use loggers inside static initialisation blocks. However, if such logging is made via another statically initialised logger reference, that statically initialised logger reference may not be initialised yet, and the application will crash. Due to this, the safe way of logging from within static initialisation blocks is done by obtaining the logger from inside the initialisation block by calling *Logger::getInstance*. This will always obtain a valid logger reference.

Logging system shutdown happens from the destructor of the statically allocated singleton. The C++ standard states that the order of destruction of statically allocated objects is non-deterministic. Logging from inside the destructors of statically allocated objects is thus not safe and should not be performed, as the logging system may have already been shut down by the time the destructor runs.

Logging messages

To log a message, the logger has five sets of templated methods that accept a variable number of parameters. Each set corresponds to a logging level { trace, debug, info, warn, error }.

The templated methods accept any types as const references. Each parameter is converted to a string via the Balau universal to-string function (see the documentation on the universal to-string function for more information). It is sufficient to ensure that a *toString* function has been defined for the type of each parameter at the point of template function instantiation, and the logger will use it to stringify the parameter.

In order for the compiler to pick up the correct *toString* functions during the function template instantiation, the header file(s) containing the function(s) must be included before the *Logger.hpp* header is included.

It is often more convenient to add an incomplete ***class Logger;*** declaration in the header file where a logger reference is defined, and include the *Logger.hpp* header in the body file instead. This ensures that verification of header include order is confined to the single body file instead of propagating to all files that include the header containing the logger reference.

The log message passed to the call should contain the same number of parameter placeholders as is passed to the function. A parameter placeholder has the form of a pair of curly braces "{}". Each placeholder is then replaced during the log call by the associated parameter.

Writing a log message thus has the following form:

```
ComplexObject results = process();

LOG.info("The results of run {}/{} are: {}", runIndex, runCount, results);
```

In order to output source code file names, file paths, and/or line numbers of log message locations, it is necessary to call the logging functions that take file path and line information via a *SourceCodeLocation* instance:

```
LOG.info(SourceCodeLocation(__FILE__, __LINE__), "blah");

LOG.info(SourceCodeLocation(__FILE__, __LINE__), "blah {} {}", one, two);

LOG.info(SourceCodeLocation(__FILE__, __LINE__), [&o1, &o2] () {
    return toString("value = ", one * two);
});
```

The *__FILE__* and *__LINE__* macros are the standard C++ preprocessor macros that provide file path and line information. They must be physically placed in the source code line.

A better alternative to explicitly specifying the file path and line macros, *Logger.hpp* contains a set of convenience macros that define the source code file paths and line macros implicitly:

```
#define BalauLogTrace(LOGGER, ...)
#define BalauLogDebug(LOGGER, ...)
#define BalauLogInfo(LOGGER, ...)
#define BalauLogWarn(LOGGER, ...)
#define BalauLogError(LOGGER, ...)
```

These are used as function calls that take the logger as the first argument and the message /parameters as subsequent arguments:

```
BalauLogInfo(LOG, "blah");

BalauLogInfo(LOG, "blah {} {}", one, two);

BalauLogInfo(LOG, [&o1, &o2] () { return toString("value = ", one * two);
```

Use of these macros is recommended in preference to the explicit calls.

If you do not wish to see the *Balau* prefix on each logging line of your code, create a set of your own macros which delegate to the above macros:

```
#define MyAppLogTrace(LOGGER, ...)  BalauLogTrace(LOGGER, __VA_ARGS__)
#define MyAppLogDebug(LOGGER, ...)  BalauLogDebug(LOGGER, __VA_ARGS__)
#define MyAppLogInfo(LOGGER, ...)   BalauLogInfo(LOGGER, __VA_ARGS__)
#define MyAppLogWarn(LOGGER, ...)   BalauLogWarn(LOGGER, __VA_ARGS__)
#define MyAppLogError(LOGGER, ...)  BalauLogError(LOGGER, __VA_ARGS__)
```

Logging namespaces

The logging framework uses hierarchical logging namespaces to determine the logging configuration for each logger. Logging namespaces are dot delimited names with the following syntax:

```
[a-zA-Z][a-zA-Z0-9_]*(\.[a-zA-Z][a-zA-Z0-9_]*)*
```

Logging namespaces work in a hierarchical manner by inheriting the configuration of the nearest ancestor unless overridden in the immediate logger namespace configuration. For example, if the logging configuration supplied during the configuration of the logging system is:

```
com.borasoftware {
    level: info
}

com.borasoftware.a {
    level: debug
}
```

then a logger instantiated with the *com.borasoftware* namespace:

```
Balau::Logger & logger = Balau::Logger::getLogger("com.borasoftware");
```

will log to info level. If a logger is instantiated with the *com.borasoftware.a* namespace:

```
Balau::Logger & logger = Balau::Logger::getLogger("com.borasoftware.a");
```

then it will log to debug level. If a logger is instantiated with the *com.borasoftware.b.c* namespace (i.e. a namespace that is not directly specified in the configuration but has ancestor *com.borasoftware*):

```
Balau::Logger & logger = Balau::Logger::getLogger("com.borasoftware.b.c");
```

then this logger will log to the level specified by *com.borasoftware*.

If a logger is instantiated with an unknown namespace with no known ancestors, the global logging namespace configuration will be used. The global namespace may also be configured in the logging configuration by using the special "." namespace.

Configuration file

Overview

The configuration file format is standard environment configuration. Each composite property defines a namespace key and a set of configuration value properties within.

The logging system can be configured via a standalone configuration file, or can be configured from the injector's loaded environment configuration by obtaining a suitable *EnvironmentProperties* instance from the injector and calling the *Logger::configure (EnvironmentProperties)* function.

In addition to the following documentation, the logging environment configuration pages provide reference documentation on each logging value property.

The global logging namespace is specified via the special "." key. This namespace can be configured in the same way as any other namespace, and forms the global ancestor to all namespaces. It is thus useful to place any configuration applicable to all loggers in the global namespace.

Here is an example of a simple logging configuration:

```
. {
    level  = warn
    format = %Y-%m-%d %H:%M:%S [%thread] %level - %namespace - %message
}

com.borasoftware {
    level = info
}

com.borasoftware.a {
    level = debug
}
```

This example configuration configures three logging namespaces. The global namespace logging level is set to warn. Two other descendant namespaces' levels are also set. All loggers will log to the default stdout/stderr streams.

The example configuration configures the global namespace with a logging message format. Unless this is reconfigured in a descendant namespace, all loggers will use this format.

Configuration macros

Logging configurations may contain the following macro placeholders in the output stream specifications. With the exception of the *${date}* placeholder, these placeholders are expanded during logging system configuration.

Placeholder	Expansion
${user.home}	The path to the user's home directory in file URI format (example: *file:///home/bob*).
${executable}	The name of the executable.
${date}	The current date.

For example, the logging system could be configured to log all output to a file contained within the user's home directory and with the same name as the executable by using the following stream option:

```
. {
    stream = ${user.home}/${executable}.log
}
```

The date placeholder

This functionality is not yet implemented.

Unlike the other placeholders, the *${date}* placeholder is not expanded before configuration parsing. Instead, the *${date}* placeholder is processed later on by the stream class.

Basic usage

When a stream URI contains the *${date}* placeholder, the logging system will automatically change the stream URI each day with the *${date}* placeholder updated to the new current date. This will result in the logging output changing at midnight each day. This can be useful when a new logging file is required each day, without requiring an application restart.

As this functionality is implemented in the *FileLoggingStream* class, custom logging stream classes are responsible for implementing this functionality if required. If a custom logging stream implementation is specified in the logging configuration, the associated custom logging stream class will need to be programmed with the necessary logic to recognise and expand the *${date}* placeholder, and close/open the relevant output streams each day.

The previous configuration example with with the *${date}* placeholder added to the file logging stream is:

```
. {
    stream = ${user.home}/${executable}-${date}.log
}
```

Date options

Date placeholders may take up to two options. These options are placed within the {} brackets of the placeholder, after the *date* placeholder keyword. The options are whitespace delimited.

The options are as follows.

Option	Description
Compress	When the "compress" option is specified, the previous logging file is compressed when the day advances.
Date format	The text format of the date used in the logging file names. When not specified, the default %F is used. Permitted flags are: aAbBcCdeFgGhjmuUVwWyY. Permitted characters between flags are '-' and '_'.

Information on the date flags is available in the HH date library's documentation. See the DateTime documentation for more details on the date library.

Configuration options

The following configuration options are currently defined.

OPTION NAME	DESCRIPTION
level	Logging level
format	Message format specification
flush	Whether to automatically flush after each message (default is to flush)
stream	Output stream specification for all logging levels
trace-stream	Output stream specification for trace logging
debug-stream	Output stream specification for debug logging
info-stream	Output stream specification for info logging
warn-stream	Output stream specification for warn logging
error-stream	Output stream specification for error logging

Logging level

The logging level option value is just the level (trace, debug, info, warn, error, none). The value text is case insensitive.

Format specification

The message format specification consists of a printf like format string that contains text and format specifiers. The available format specifiers are:

Specifier	Description
%Y	the year as four digits
%y	the year as two digits
%m	the month as two digits
%d	the day of the month
%H	the hour as two digits
%M	the minute as two digits
%S	the seconds as two digits followed by six digits representing the microsecond remainder
%thread	the thread name if one has been set or the thread id otherwise
%level	the logging level in lowercase
%LEVEL	the logging level in uppercase
%namespace	the logger's logging namespace
%ns	an abbreviation of the logger's logging namespace, created by printing each identifier's first letter only, apart from the last identifier which is printed in its entirety
%filename	the source code file name
%filepath	the full path to the source code file
%line	the line number in the source code file
%message	the message, after stringification and combination of all arguments
%%	the percent character
%"	the double quotation character

For the *%thread* specifier, the thread name can be set by calling *Util::ThreadName::setName (name)* from the thread. The *Util::ThreadName* class is a utility that stores a thread-local name that is used by the logging system for the purpose of replacing the thread id with a meaningful name. Note that currently, the name of a thread can only be set from within the thread itself.

The default format specification if none is supplied or inherited for a particular namespace is:

```
%Y-%m-%d %H:%M:%S [%thread] %LEVEL - %namespace - %message
```

An alternative to the default logging format that includes the filename and line number could be:

```
%Y-%m-%d %H:%M:%S %filename:%line - [%thread] %LEVEL - %namespace - %messag
```

Log messages resulting from this format would be similar to the following:

```
2018-03-26 18:19:59.904675835 LoggerTest.cpp:139 - [main] INFO  -  - hello
2018-03-26 18:19:59.904699204 LoggerTest.cpp:140 - [main] INFO  - com.boras
```

Flush

The flush option allows automatic flushing to be selectively disabled for different logging namespaces.

By default, loggers automatically flush the writer stream after writing a logging line. This ensures that logging is physically written to output streams promptly. Automatic flushing can be disabled for a namespace by specifying *flush: false* in the namespace configuration.

Stream specifications

The stream specification options specify the output stream(s) to be created and written to for the logging namespace. The value of the options is a URI:

```
stream          = [uri]
[level]-stream = [uri]
```

where *[uri]* is a real or pseudo URI identifying a stream to write to. Stream URIs are supported by logging system plugins.

URIs that are handled by default are:

- standard localhost file URLs (file:///path/to/file);
- stdout/stderr file descriptor pseudo schemes.

Built in stream configuration examples:

```
stream       = file:///path/to/file
warn-stream  = stdout
error-stream = stderr
```

Output streams for other types of URI are instantiated by logging system plugins (see next section).

The file descriptor pseudo schemes log to the application's standard output and error streams.

For a particular stream specification, the logging system will instantiate an output stream if the stream type is recognised by the logging system or one of the registered plugins. If a stream specification is not recognised, the logging system will configure the logger to the null output stream and will log an error message to the output stream of the global logging namespace.

The different stream specification options correspond to non-level specific and level specific stream configurations. The non-level specific *stream* option sets a single stream for all logging levels. All log output for a particular namespace will log to this stream when this option is specified with no other stream options.

The level specific *[level]-stream* options allow different output streams to be configured for different logging levels. Each level specific stream option configures the log output of that level.

When one or more level specific *[level]-stream* options are specified and the non-level specific *stream* option is not specified, the streams of levels that do not have any stream specification will be determined via the stream configurations of adjacent levels. In this case, the priority of the adjacent level specific stream configurations is first downwards, then upwards. For example, if the info level is configured via a level specific stream configuration to log to stdout and the error level is configured via a level specific stream configuration to log to stderr, the warn level will be implicitly configured to log to stderr and the trace and debug levels will be implicitly configured to log to stdout.

If the non-level specific *stream* option is specified along with one or more level specific *[level]-stream* options, all levels without level specific *[level]-stream* options will be configured to the non-level specific stream setting.

Some additional examples will make this concept more clear:

```
com.borasoftware {
    warn-stream  = stdout
    error-stream = stderr
}
```

The above specification will configure the com.borasoftware logger to output all trace, debug, info, and warn level log messages to stdout, and all error level log messages to stderr.

```
com.borasoftware {
    debug-stream = ${user.home}/.bora/${executable}/debug.log
    warn-stream  = stdout
    error-stream = stderr
}
```

The above specification will configure the com.borasoftware logger to output all trace and debug level log messages to the specified file, all info and warn level log messages to stdout, and all error level log messages to stderr.

If, for example the user home directory is */home/bob* and the executable name is *BalauTests* , the resulting debug stream for the above example would be *file:///home/bob/.bora /BalauTests/debug.log*.

```
com.borasoftware {
    stream       = stdout
    error-stream = stderr
}
```

The above specification will configure the com.borasoftware logger to output all logging to stdout apart from the error stream, which will be configured to output to stderr.

Logging stream plugins

Additional logging streams may be registered with the logging system before reconfiguring the system with schemes referencing these custom logging streams. This is achieved by deriving a new class from the *LoggingStream* base class, and creating a factory function with the same signature as specified by the *LoggingStreamFactory* typedef.

The *LoggingStream* base class has the following virtual methods:

```
class LoggingStream {
    public: virtual void write(const std::string & str) = 0;
    public: virtual void flush() = 0;
};
```

The string passed to the write method is the pre-formatted message.

The signature for logging stream factory functions is:

```
using LoggingStreamFactory = LoggingStream * (*)(const std::string & uri);
```

The logging system is a pointer container for the logging streams, thus the logging stream factories return a raw pointer instead of a pointer container.

The URI passed to logging stream factory functions is the URI from the logging configuration text, with all macro placeholders expanded apart from the *${date}* placeholder. If the URI specified in the logging configuration contains one or more occurrences of the *${date}* macro placeholder, this placeholder will be present in the URI .

The factory function for the custom logging stream needs to be registered with the logging system by calling:

```
Logger::registerLoggingStreamFactory(const std::string & scheme, LoggingSt
```

The uri string of the custom logging stream is not specified by the logging system. The custom logging stream class must parse the supplied uri with its own DSL.

Design

This section provides a summary of the design of the logging system. It is not necessary to read this section in order to use the logging system.

Overview

The logging system is based around the public *Logger* class and the private *LoggingState* class. The *LoggingState* class is instantiated as a lazy singleton. The logging state contains a tree of loggers, plus pools for logging streams, log items, and composed log item vectors. The logging system contains a mutex that is locked when a logger is requested and when a reconfiguration is performed. Logging does not lock the mutex, hence concurrent logging and reconfiguration is supported.

Logging configuration text supplied to the logging system is parsed via the standard hierarchical properties parser.

The logging system is configured at application startup, in the first call to either *Logger::getInstance* or *Logger::configure*.

If a call to *Logger::getInstance* is the first call, the following procedure occurs:

1. a default logging configuration is created;
2. an override logging configuration is created from the contents of the *balau-logging.hconf* file if it exists;
3. the logging configuration created from *balau-logging.hconf* (if it exists) is cascaded onto the default logging configuration;
4. the properties of each resulting parent logger are propagated onto the children;
5. the logging levels of the loggers are set from the resulting level properties;

6. the streams of the loggers are set from the stream properties;
7. the message format item vectors of the loggers are set from the message format properties.

If a call to *Logger::configure* is the first call or on any subsequent call to *Logger::configure*, the following procedure occurs:

1. a default logging configuration is created;
2. an override logging configuration is created from the contents of the string passed to the *configure()* method;
3. the new logging configuration is cascaded onto the default logging configuration;
4. the properties of each resulting parent logger are propagated onto the children;
5. the properties of the existing loggers are wiped, ready for the new properties;
6. the logging levels of the loggers are set from the resulting level properties;
7. the streams of the loggers are set from the stream properties;
8. the message format item vectors of the loggers are set from the message format properties.

Concurrency

Configuration of the logging system is covered by a mutex. Only a single configuration execution may take place at any one time. The mutex is not locked on normal logging calls.

During configuration or reconfiguration, all missing streams, log items, and log item vectors are created and pooled. Subsequently, the atomic fields of new or existing loggers are set. This two stage process allows concurrent configuration and logging to occur without invalidating any of the existing loggers' state. In addition, the pooling ensures that there is no duplication of identical logging state, minimising the memory requirements of the logging system when reconfigured.

Due to *TSO* memory ordering, the atomic reads performed when logging are free reads on x86/x86-64 platforms.

The *Logger* class has the following fields that are read when logging a message. All mutable fields are implemented as atomic fields.

Field name	Comments
namespace	Doesn't change for the lifetime of the logger.
ns	Doesn't change for the lifetime of the logger.
level	Logging levels are read with *std::memory_order_relaxed* semantics.
stream[x]	The set of stream pointers is kept within a *std::array<std::atomic<LoggingStream *>, 5>*. Pointers are read with *std::memory_order_relaxed* semantics.
loggerItems	The entire log item vector is iterated over when logging a message. A pointer to the vector is read before iteration. The pointer is read with *std::memory_order_acquire* semantics.

All mutable logger state is updated with *std::memory_order_seq_cst* semantics during reconfiguration.

Test runner

Overview

A unit testing framework, allowing tests to be defined as member functions in test group classes. The test runner contains four different execution models, allowing single threaded and concurrent runs, in and out of process.

Unlike many other C++ unit test frameworks, the Balau unit test framework does not use preprocessor macros. Instead, tests are defined as parameterless instance methods of test group classes, and assertions are Hamcrest inspired template functions. A complete test class forms a test group in the resulting run. Test classes linked into the test application are automatically instantiated and register themselves with the test framework.

The Balau unit test framework does not use any external code generation tool to simulate the effects of the Java annotations in Java based unit test frameworks such as JUnit and TestNG, from which many C++ unit test frameworks are inspired. Instead, the test methods of a class are simply added to the class' test run by specifying them within the constructor.

Test classes typically mirror the production code classes. Using a friend class declaration in a production class allows the test class' methods to test private functions if this is required.

The Balau test runner has four execution models. The models are:

- single process, single threaded;
- single process, multi-threaded;
- multiple worker process;
- separate process per test.

The execution model can be selected by passing the relevant execution model enum value to the test runner initialisation method from the test application main function.

All tests are run by default. Selective running of tests is achieved by passing test group / test case names to the test runner initialisation method, via the *argc* and *argv* parameters in the test application's main function. Simple globbing can also be used to specify multiple tests to run.

Quick start

#include <Balau/Testing/TestRunner.hpp>

Test groups

Tests are defined within test groups. Each test group is defined as a class.

Test group classes inherit from the *Testing::TestGroup* test runner base template class, using CRTP. Each test case is an instance method of the test class, which takes zero parameters and returns void. The constructor body of a test class registers test methods via calls to the *registerTestCase* method or the *RegisterTestCase* convenience macro.

```
// Example test group from the Balau unit tests.

struct ObjectTrieTest : public Testing::TestGroup<ObjectTrieTest> {
    ObjectTrieTest() {
        RegisterTestCase(uIntTrieBuild);
        RegisterTestCase(uIntTrieCopy);
        RegisterTestCase(uIntTreeDepthIterate);
        RegisterTestCase(uIntTreeDepthIterateForLoop);
        RegisterTestCase(uIntTreeBreadthIterate);
        RegisterTestCase(fluentBuild);
    }

    void uIntTrieBuild();
    void uIntTrieCopy();
    void uIntTreeDepthIterate();
    void uIntTreeDepthIterateForLoop();
    void uIntTreeBreadthIterate();
    void fluentBuild();
};
```

Test application

The test application executes the test runner by calling the test runner's *run* method. Within the main function, the test runner is initialised via one of the runner's initialisation methods. The most common *run* methods to use are the ones that take *argc* and *argv* arguments. The *argc* and *argv* arguments of the test application's main function are passed to the test runner's *run* method, in order to parse the command line arguments.

```
#include <Balau/Testing/TestRunner.hpp>

using namespace Balau::Testing;

int main(int argc, char * argv[]) {
    return TestRunner::run(argc, argv);
}
```

The test runner uses the command line parser to parse a space separated command line which can also contain globbed test name patterns to run.

The following command line options are available.

Short option	Long option	Has value	Description
-e	--execution-model	yes	The execution model (default = SingleThreaded).
-n	--namespaces	no	Use namespaces in test group names (default is not to use).
-p	--pause	no	Pause at exit (default is not to pause).
-c	--concurrency	yes	The number of threads or processes to use to run the tests (default = detect).
-r	--report-folder	yes	Generate test reports in the specified folder.
-h	--help	no	Displays this help message.

The test runner will interpret the first element of *argv* as the execution model (case insensitive) if it is a valid execution model, and the remainder of the command line arguments as a space/comma delimited list of globbed test names to run. If the first element of *argv* is not a valid execution model, it will form the head of the globbed test name list and the *SingleThreaded* execution model will be used by default.

In both cases, the zeroth element of *argv* is assumed to be the test application path and is ignored. If this is not the case, then the starting element can be specified as an optional argument of the *run* call.

Selecting tests

Selective running of test cases is achieved by providing a space/comma delimited list of globbed test names to the test runner's *run* method. If no list is provided, all test cases are run.

There are two globbing patterns available:

Glob character	Meaning
*	Match zero or more characters.
?	Match exactly a single character.

Multiple patterns can be specified on the command line, either with a single command line argument containing a comma delimited list, or via multiple command line arguments representing a space delimited list.

```
# Run the Balau test application with the worker processes execution model
# and specifying a subset of tests via a comma delimited list of patterns.
BalauTests -e WorkerProcesses Injector::*,Environment::*

# Run the Balau test application with the worker processes execution model
# and specifying a subset of tests via a space delimited list of patterns.
BalauTests -e WorkerProcesses Injector::* Environment::*
```

Execution models

The test runner has four execution models. The models are:

- single process, single threaded;
- single process, multi-threaded;
- worker process;
- process per test.

For the multi-threaded and worker process execution models, the concurrency level (the number of threads or processes) can be optionally specified as an argument to the test runner constructor. The concurrency level is also used to specify the number of simultaneous processes to spawn in the process per test execution model.

If the concurrency level is not specified, the default value equal to the number of CPU cores is used.

Defining tests

Test groups

Tests are grouped inside test classes.

Test classes derive from the *Testing::TestGroup* base class template, using CRTP. Each test is an instance method of the test class, which takes zero parameters and returns void. The constructor body of a test class registers test methods via calls to the *registerTestCase* method or the *RegisterTestCase* convenience macro.

The following is the header of a test class which has four test methods.

```cpp
struct CommandLineTest : public Testing::TestGroup<CommandLineTest> {
    CommandLineTest() {
        RegisterTestCase(basicTest);
        RegisterTestCase(failingTest);
        RegisterTestCase(finalValueTest);
        RegisterTestCase(numericValueTest);
    }

    void basicTest();
    void failingTest();
    void finalValueTest();
    void numericValueTest();
};
```

The result of running the above test class follows. One of the tests is failing.

```
------------------------- STARTING TESTS -------------------------

Run type = single process, multi-threaded (2 threads)

++ Running test group CommandLineTest

 - Running test CommandLineTest::basicTest         - passed. Duration = 17
 - Running test CommandLineTest::failingTest       - FAILED!

Assertion failed:
true != false

 Duration = 140us

 - Running test CommandLineTest::finalValueTest    - passed. Duration = 14
 - Running test CommandLineTest::numericValueTest  - passed. Duration = 35

== CommandLineTest group completed. Group duration (core clock time) = 675

------------------------- COMPLETED TESTS ------------------------

Total duration   (test run clock time)    = 675µs
Average duration (test run clock time)    = 225µs
Total duration   (application clock time) = 696µs

THERE WERE TEST FAILURES.

Total tests run: 4

  3 tests passed
  1 test failed

Failed tests:
    CommandLineTest::failingTest

Test application process with pid 13090 finished execution.
Process finished with exit code 0
```

Setup and teardown

Test classes may include setup and teardown methods. Due to the multi-process design of the test runner, only test setup/teardown methods are supported (i.e. there are no class setup/teardown methods included).

The following is a test class which has test setup and teardown methods defined:

```
class CommandLineTest : public Testing::TestGroup<CommandLineTest> {
    public: CommandLineTest() {
        RegisterTestCase(basicTest);
        RegisterTestCase(finalValueTest);
        RegisterTestCase(numericValueTest);
    }

    void basicTest();
    void finalValueTest();
    void numericValueTest();

    private: void setup() override {
        log("      CommandLineTest::setup() called.\n");
    }

    private: void teardown() override {
        log("      CommandLineTest::teardown() called.\n");
    }
};
```

The result of running the above test class follows.

```
------------------------- STARTING TESTS -------------------------

Run type = single process, single threaded

++ Running test group CommandLineTest

 - Running test CommandLineTest::basicTest         - passed. Duration = 65
      CommandLineTest::setup() called.
      CommandLineTest::teardown() called.
 - Running test CommandLineTest::finalValueTest    - passed. Duration = 47
      CommandLineTest::setup() called.
      CommandLineTest::teardown() called.
 - Running test CommandLineTest::numericValueTest  - passed. Duration = 19

== CommandLineTest group completed. Group duration (core clock time) = 303

------------------------- COMPLETED TESTS ------------------------

Total duration   (test run clock time)    = 303µs
Average duration (test run clock time)    = 101µs
Total duration   (application clock time) = 393µs

ALL TESTS PASSED: 3 tests executed
Test application process with pid 13431 finished execution.
Process finished with exit code 0
```

Assertions

The test runner contains Hamcrest inspired assertion utilities that are available for use in test methods and setup/teardown methods.

In order to use the assertions, suitable ***operator ==*** functions/methods will need to exist for the types of the objects specified in the assertion statements. In addition, each type will require a *toString* function to be defined. These are used by the assertion render functions. Refer to the documentation on the universal to-string function for more information.

In order for the compiler to pick up the correct *toString* and *operator ==* functions, the header file(s) containing the functions must be included before the *TestRunner.hpp* header is included.

When writing tests in a *.cpp* file, it is convenient to import the assertion function symbols via a using directive:

```
// Import the assertion functions.
using namespace Balau::Testing;
```

Examples of assertions can be found in the *AssertionsTestData.hpp* test file.

There are two ways to use the assertion functions. The first is directly:

```
Balau::Testing::assertThat(actual, is(expected));
```

When the assertion fails, the assertion will log an error by calling *toString* on each of the arguments and then throw a *Balau::Exception::AssertionException*.

The alternative and recommended way of using the assertion functions is via the *AssertThat* macro. This macro also performs the assertion via Balau::Testing::assertThat, but in addition to the assertion, the *__FILE__* and *__LINE__* macros are used in order to supply the source code location to the assertion function for logging.

```
AssertThat(actual, is(expected));
```

As the *AssertThat* token is a macro, it should not be prefixed by a namespace.

Comparisons

Assertions for equality and other standard comparisons are available:

```
AssertThat(actual, is(expected));
AssertThat(actual, isNot(expected));
AssertThat(actual, isGreaterThan(expected));
AssertThat(actual, isLessThan(expected));
AssertThat(actual, isGreaterThanOrEqual(expected));
AssertThat(actual, isLessThanOrEqual(expected));
AssertThat(actual, isAlmostEqual(expected, errorLimit));
```

These comparison assertions require that the implicated types have corresponding comparison functions defined. For the *isAlmostEqual* assertion, both *<=* and *>=* are required.

Other types of comparison such as startsWith, endsWith, etc. are also available:

```
AssertThat(actual, startsWith(expected));
AssertThat(actual, endsWith(expected));
AssertThat(actual, contains(expected));
AssertThat(actual, doesNotContain(expected));
```

However, these assertions require that the type implicated in the call have a *std::basic_string* type API (length, substr, begin, end) and require the *Balau::contains(actual, expected)* function to be defined for the actual and expected types, so unless you define such an API and helper function for your types, their use is limited to *std::basic_string<T>*.

Exceptions

Assertions for expected thrown exceptions are available with a similar Hamcrest like API:

```
AssertThat(function, throws<T>());
AssertThat(function, throws(expectedException));
AssertThat(function, throws(expectedException, comparisonFunction));
```

The usage of these assertions differs from the comparison assertions. The first argument passed to the *assertThat* call is a function, typically supplied as a lambda:

```
AssertThat([&] () { foo(); }, throws<T>());
AssertThat([&] () { foo(); }, throws(expectedException));
AssertThat([&] () { foo(); }, throws(expectedException, comparisonFunction
```

The function is called during the assertion call, and any exception thrown is then examined. The first call verifies that the expected type of exception is thrown. The second and third calls examine the contents of the thrown exception, compared to the supplied exception. In order to use the second call, the exception type must have an equality operator function defined for it in order for the code to compile. The third call allows a comparison function to be passed to the call, which will be used instead of the equality operator.

In order to use the exception instance assertion versions, a suitable *operator ==* function must be defined for the exception, in order that the test framework may compare the actual and expected exceptions. No such function is required in order to use the exception type assertion version. A suitable *toString* function will also be required for the exception class, in order for the test runner to print the exception contents during assertion failures.

Examples of the use of the first and third exception assertion calls can be seen in the *CommandLineTest* class:

```
// Exception type assertion.
AssertThat([&] () { commandLine.getOption(KEY9); }, throws<OptionValueExce

// Exception contents assertion.
auto comp = [] (auto & a, auto & e) { return std::string(a.what()) == std:
AssertThat([&] () { commandLine.getOption(KEY9); }, throws(OptionValueExce
```

Renderers

The assertion methods call the following function to print the content of the actual and expected values in the case of an assertion failure:

```
namespace Balau {

namespace Renderers {

template <typename A, typename E>
std::string render(const A & actual, const E & expected);

} // namespace Renderers

} // namespace Balau
```

The standard renderer calls the Balau *toString* functions for the inputs and prints each resulting line side by side, along with an infix *==* or *!=* according to line equality.

Custom failure message renderers may be added by specialising the above template function (within the Balau namespace):

Logging

Test output

The test runner has configurable test logging, allowing the test log to be written to a variety of outputs. This is achieved by passing one or more *TestWriter* derived classes to the test runner's *run* method. The following test writers are defined in the *TestRunner.hpp* header:

Class name	Description
StdOutTestWriter	Write to stdout.
FileTestWriter	Write to the specified file.
OStreamTestWriter	Write to the previously constructed output stream.
LogWriter	Write to the specified Balau logger.

Other test writers may be created if required, by deriving from the *TestWriter* base class.

In order to register test writers, they are specified as arguments to the test runner's *run* method:

```
int main(int argc, char * argv[]) {
    // Run the test runner with two writers.
    TestRunner::run(
          argc, argv, 1, false, false
        , LogWriter("balau.test.output")
        , FileTestWriter(Resource::File("testOutput.log"))
    );
}
```

If no test writers are specified, the test runner will log to stdout by default.

Test logging

The *TestGroup* base class contains two logging methods that can be used to log output to the test runner writers:

```
///
/// Write additional logging to the test writers.
///
protected: void log(const std::string & string);

///
/// Write additional logging to the test writers.
/// A line break is written after the string.
///
protected: void logLine(const std::string & string = "");
```

These methods can be used anywhere in a test class to log additional test messages.

An alternative technique of logging test messages is to use the Balau logging framework and construct the test runner to log test results to a Balau logger. This allows the full parameter parsing of the logging framework to be used within the test class log messages. There are two ways of configuring the logging system for the test application.

1. Implicitly configure the logging system with a *balau-logging.hconf* file in the test application binary directory (a CMake custom command will be required to copy the logging configuration file to the build folder - see the Balau CMakeLists.txt for an example).
2. Explicitly configure the logging system by calling *Logger::configure(std::string)* from within the test application main function, before running the tests.

More information on logging system configuration is available in the Logging documentation.

Test reports

In addition to test result logging, the test runner can be configured to generate XML based test run report files.

The default reporter provides Maven Surefire plugin schema based XML reports. Alternative report generators may be defined by deriving from the *TestReportGenerator* base class and providing an instance of the reporter class to the test runner *run* function.

Test utilities

The test framework utilities are designed to provide domain specific help in accomplishing certain management tasks required during testing. The test framework utilities are in an early stage of development, and currently only a pair of network related test utility functions are defined.

Network

The test framework network utilities provide a way of getting a free TCP ports for tests. There are two functions defined:

- initialiseWithFreeTcpPort;
- getFreeTcpPort.

These two functions are normally used together. The *initialiseWithFreeTcpPort* function takes some code to execute. This code is part of the test and should initialise the network state that requires the port.

From within the code supplied to the *initialiseWithFreeTcpPort* function, the *getFreeTcpPort* function should be called in order to obtain a free port that will be used to initialise the network state.

There are two possible failures that may occur when attempting to obtain a free port. The first is that the specified port is not available. This issue is mitigated with the call to *getFreeTcpPort*. This function takes start port and port count arguments, and tests for the availability of a free port between the specified port range.

Once a free port has been obtained, an attempt to bind to it can be made by the test code. However, there is an inherent race condition present. If another process binds to the port between the call to *getFreeTcpPort* and the subsequent attempt to bind, the binding will fail.

In order to mitigate this race condition, the *initialiseWithFreeTcpPort* function will repeatedly run the initialisation code until a successful binding has been achieved. Each time the initialisation code is run, a new free port is obtained via the *getFreeTcpPort* call.

Examples of this pattern being used can be seen in the HTTP network tests in the Balau test suite. The following is an extract from one of the tests in the *FileServingHttpWebAppTest* class.

```
const unsigned short port = Testing::NetworkTesting::initialiseWithFreeTcpl
    [&server, documentRoot, testPortStart] () {
        auto endpoint = makeEndpoint(
            "127.0.0.1", Testing::NetworkTesting::getFreeTcpPort(testPortS
        );

        auto clock = std::shared_ptr<System::Clock>(new System::SystemClocl

        server = std::make_shared<HttpServer>(
            clock, "BalauTest", endpoint, "FileHandler", 4, documentRoot
        );

        server->startAsync();
        return server->getPort();
    }
);
```

In the above code, the race condition occurs between the call to *getFreeTcpPort* and the call to construct the HTTP server.

Test application

Main function

All test group classes that are linked into the test application are automatically instantiated and registered with the test runner. The *main* function of the test application should thus only call one of the *run* methods of the test runner.

Example *main* function:

```
#include <Balau/Testing/TestRunner.hpp>

using namespace Balau::Testing;

int main(int argc, char * argv[]) {
    return TestRunner::run(argc, argv);
}
```

Selecting tests

Selective running of test cases is achieved by providing a space/comma delimited list of globbed test names to the test runner's *run* method. If no list is provided, all test cases are run.

There are two globbing patterns available:

Glob character	Meaning
*	Match zero or more characters.
?	Match exactly a single character.

Multiple patterns can be specified on the command line, either with a single command line argument containing a comma delimited list, or via multiple command line arguments representing a space delimited list.

```
# Run the Balau test application with the worker processes execution model
# and specifying a subset of tests via a comma delimited list of patterns.
BalauTests -e WorkerProcesses Injector*,Environment*

# Run the Balau test application with the worker processes execution model
# and specifying a subset of tests via a space delimited list of patterns.
BalauTests -e WorkerProcesses Injector* Environment*
```

Model selection

TODO update documentation to reflect command line parsing

The test execution model to run can be specified either as the first argument of the command line by calling the *TestRunner::run(argc, argv)* method, or explicitly by using other *run* method overloads, most commonly the *TestRunner::run(model, argc, argv)* overload.

```
#include <Balau/Testing/TestRunner.hpp>

using namespace Balau::Testing;

int main(int argc, char * argv[]) {
    // Run the tests with the worker processes execution model.
    return TestRunner::run(WorkerProcesses, argc, argv);
}
```

The *TestRunner::run(argc, argv)* approach can be useful for teams that run a continuous integration server that requires a *WorkerProcesses* or *ProcessPerTest* execution model, whilst allowing developers to set a *SingleThreaded* or *WorkerThreads* execution model in their run configurations.

If the *TestRunner::run(argc, argv)* method is used and no command line arguments are supplied, the default *SingleThreaded* execution model is used.

Execution models

The test runner has four execution models. The models are:

- single process, single threaded;
- single process, multi-threaded;
- worker process;
- process per test.

Each execution model has advantages and disadvantages. Typically, the single process execution models would be used whilst running tests on the developer's workstation, and one of the multiple process execution models would be used when running on a continuous integration server.

Single threaded

The single threaded execution model executes each test in turn, within the test application main thread. This is the simplest of the execution models. If a test causes a segmentation fault, the test application will terminate.

Reasons to use the single threaded execution model include:

- ensures any possible complexity introduced by the test framework is eliminated;
- an in-process execution model allows direct debugging.

The disadvantages of the single threaded execution model are:

- if a test causes a segmentation fault, the test application will terminate;
- due to the single threaded nature of this execution model, test runs will take longer than with the other execution models.

Multi-threaded

The multi-threaded execution model executes tests in parallel, in a fixed number of worker threads in the test application. Each worker thread executes by claiming the next available test and running it. As this execution model is also single process, the test application will terminate if a test causes a segmentation fault.

Reasons to use the multi-threaded execution model include:

- an in-process execution model allows direct debugging;
- running tests in parallel uses all CPU cores and ensures a faster test run.

The disadvantage of the multi-threaded execution model is that if a test causes a segmentation fault, the test application will terminate.

Worker process

The worker process execution model executes tests in parallel, in a fixed number of child processes spawned by the test application. Each worker process executes by claiming the next available test and running it.

As this execution model is multiple process, the test application will not terminate if a test causes a segmentation fault. Instead, the child process will terminate and the parent process will spawn a replacement child process to continue testing.

Reasons to use the worker process execution model include:

- tests that cause segmentation faults will not result in test application termination;
- running tests in parallel uses all CPU cores and ensures a faster test run.

The disadvantage of the worker process execution model is that breakpoints will not be hit in the child processes.

Process per test

The process per test execution model executes tests in parallel, by forking a new child process for each test.

As this execution model is multiple process, the test application will not terminate if a test causes a segmentation fault. Instead, only the child process for the test causing the segmentation fault will terminate.

Reasons to use the process per test execution model include:

- the entire process' state is reset for each test;
- tests that cause segmentation faults will not result in test application termination;
- running tests in parallel uses all CPU cores and ensures a faster test run.

The disadvantages of the process per test execution model are:

- breakpoints will not be hit in the child processes;
- the forking of a new child process for each test may possibly cause a reduction in performance compared to the worker process execution model (this does not appear to be the case on x86-64 Linux).

Performance

The following test run timing information was obtained by running the Balau unit tests (2018.9.1 release) for each execution model. The CPU used in the test was an Intel i7-8550U (4 core with hyper-threading), with turbo turned off. The default concurrency of 8 threads/processes was used for the multi-threaded, worker process, and process per test execution models.

The best result of 10 runs was taken for each execution model. The timing values indicate clock time, thus they do not take into account context switches where a CPU core is executing other application's background code. The total duration (test run clock time) is the sum of all the test's execution times. For concurrent execution models, this is spread across the allocated cores. The average duration is the test run clock time total duration divided by the number of tests run. The total duration (application clock time) is the duration of the test application's main process.

In the Balau test suite, the logger tests are automatically disabled during multi-threaded and worker process runs. Consequently, these test groups were manually disabled for all the the runs in this performance evaluation. In addition, all test groups that involve network functionality were also disabled in order to avoid network latency issues skewing results.

```
----------------- SINGLE THREADED ------------------

Total duration   (test run clock time)    = 1.1s
Average duration (test run clock time)    = 4.2ms
Total duration   (application clock time) = 1.1s

***** ALL TESTS PASSED - 262 tests executed *****

------------------ MULTI-THREADED ------------------

Total duration   (test run clock time)    = 1.5s
Average duration (test run clock time)    = 5.6ms
Total duration   (application clock time) = 345.8ms

***** ALL TESTS PASSED - 262 tests executed *****

----------------- WORKER PROCESSES -----------------

Total duration   (test run clock time)    = 1.5s
Average duration (test run clock time)    = 5.8ms
Total duration   (application clock time) = 384.0ms

***** ALL TESTS PASSED - 262 tests executed *****

----------------- PROCESS PER TEST -----------------

Total duration   (test run clock time)    = 1.2s
Average duration (test run clock time)    = 4.7ms
Total duration   (application clock time) = 382.7ms

***** ALL TESTS PASSED - 262 tests executed *****
```

It can be seen that the single threaded executor has the lower overhead per test, but the other executors nevertheless reduce the execution time significantly when run on a multi-core CPU. If an appreciable amount of tests with significant I/O waits were present, the speed up would approach the number of allocated cores.

The Balau unit test execution times range from microseconds to tens of milliseconds, and there were 262 tests run in the above timing runs. A complex C++ application could have one or two orders of magnitude more tests than this, and it is likely that the average execution time of the tests would be greater on average. Thus the overall execution times presented above could thus be multiplied by a factor of several orders of magnitude in order to represent a real world scenario.

CI configuration

This section only discusses continuous integration via a CMake target.

In order to create CI jobs in tools such as Jenkins, the test application will require a CMake target to run. This can be achieved with the following configuration in the CMakeLists.txt file.

```
##################### TEST RUNNER #####################

add_custom_target(
    RunTests
    ALL
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/bin
    COMMAND TestApp
)

add_dependencies(RunTests TestApp)
```

This custom target can also be used directly from the command line if required, by typing *make RunTests*.

In the CMake custom target definition, *RunTests* is the name of the target that will be specified in the CI jobs, and *TestApp* is the test application executable target. After adding the dependency declaration, running the *RunTests* target will first build the test application, then will execute it.

The test application will return 0 on success and 1 on failure. This will be picked up by the CI job runner in order to determine test run success.

Characters and strings

Overview

The Balau library has been designed primarily to work with external character data encoded in UTF-8 and internal character data encoded in UTF-8 and UTF-32. UTF-8 is used for persisted strings, data transfer, and in-memory strings. UTF-32 is used in code point processing algorithms that require a fixed size code point type. This allows a normally compact representation in memory, in transit, and in storage, but provides a fixed width character type for processing when required.

The C++ language *char* character type is used for UTF-8 data and the *char32_t* character type is used for UTF-32 data. As the size of the C++ language *wchar_t* character type is not defined in the specification, the *wchar_t* character type and associated *std::wstring* string type are not used in Balau.

The *char16_t* character type is not used in Balau components. A set of universal to-string and from-string function overloads is however included for UTF-16 string generation and conversion. These functions provide to-string and from-string conversions when another library works with UTF-16 strings or when application code requires UTF-16 encoded strings.

Balau uses the ICU library for unicode support. Unlike ICU, Balau uses the standard *char32_t* primitive type for representing UTF-32 characters. This is implicitly cast to and from ICU's *UChar32* (which is a *signed int*) within the Character functions.

The primary character and string related functionality that Balau provides is:

- UTF-8 and UTF-32 character utilities;
- universal to-string functions for UTF-8, UTF-16, and UTF-32 string conversion;
- universal from-string functions for UTF-8, UTF-16, and UTF-32 string conversion;
- various UTF-8 and UTF-32 string utilities;
- byte based resource classes for reading and writing UTF-8 data;
- UTF-8 to UTF-32 converting resource classes for reading UTF-32 data;
- UTF-32 to UTF-8 converting resource classes for writing UTF-32 data.

The character utilities, universal to-string and universal from-string functions are discussed in this section. The string utilities section discusses the string utilities. The resources section discusses the various resource classes.

String types

The Balau C++ library uses the following character and string types:

Char type	String type	Usage
char	std::string	UTF-8 string or undefined array of bytes
char16_t	std::u16string	UTF-16 string
char32_t	std::u32string	UTF-32 string

Character utilities

#include <Balau/Type/Character.hpp>

Character utility functions for the following themes are provided:

- classification (UTF-32);
- iteration (UTF-8);
- mutation (UTF-8 / UTF-32).

Many of the character utility functions are proxies to corresponding ICU functions.

Classification

The classification functions each accept a *char32_t* character. Most of the classification functions act as predicates.

The following predicate classification functions are available.

Function name	Description
isLower	Does the specified code point have the general category *Ll* (lowercase letter).
isUpper	Does the specified code point have the general category *Lu* (uppercase letter).
isDigit	Does the specified code point have the general category *Nd* (decimal digit numbers).
isHexDigit	Does the specified code point have the general category *Nd* (decimal digit numbers) or is one of the ASCII latin letters a-f or A-F.
isOctalDigit	Is the specified code point one of the ASCII characters 0-7.
isBinaryDigit	Is the specified code point one of the ASCII characters 0-1.
isAlpha	Does the specified code point have the general category *L* (letters).
isAlphaOrDecimal	Does the specified code point have the general category *L* (letters) or *Nd* (decimal digit numbers).
isControlCharacter	Is the specified code point a control character.
isSpace	Is the specified code point a space character (excluding CR / LF).
isWhitespace	Is the specified code point a whitespace character.
isBlank	Is the specified code point a character that visibly separates words on a line.
isPrintable	Is the specified code point a printable character.
isPunctuation	Does the specified code point have the general category *P* (punctuation).
isIdStart	Does the specified code point have the general category *L* (letters) or *Nl* (letter numbers).
isIdPart	Is the specified code point valid as part of an Id.
isBreakableCharacter	Is the specified code point a breakable character for line endings.
sInclusiveBreakableCharacter	Is the specified code point a breakable character for line endings that should be printed.

The following non-predicate classification functions are available.

Function name	Description
utf8ByteCount	Returns the number of bytes that the character occupies when UTF-8 encoded.

Iteration

Iteration functions are defined for UTF-8 string views. These functions advance or retreat an integer offset to the next or previous UTF-8 character. Two of the functions also return the resulting character.

The following iteration functions are currently available.

Function name	Description
getNextUtf8	Get the next code point from the UTF-8 string view.
getPreviousUtf8	Get the previous code point from the UTF-8 string view.
advanceUtf8	Advance the supplied offset from one code point boundary to the next one.
retreatUtf8	Retreat the supplied offset from one code point boundary to the previous one.

Mutation

Mutation functions are available for *char32_t* characters and for UTF-8 *char* characters at offsets inside *std::string* strings.

The following mutating functions are currently available.

Function name	Description
toUpper(char32_t)	Convert the supplied code point to uppercase.
toLower(char32_t)	Convert the supplied code point to lowercase.
setUtf8AndAdvanceOffset(std::string & destination, int & offset, char32_t c)	Write a code point into the supplied UTF-8 string.

Universal to-string

#include <Balau/Type/ToString.hpp>

This section outlines a development approach and supporting code in the Balau library for a universal to-string function for each of the supported unicode encoding string types. These functions are used throughout the Balau library and will propagate to application code through the Balau header files. The implementation allows application developers to define additional to-string function overloads for their own types and any other types for which they require custom to-string function implementations.

Overview

The C++ standard library provides a *to_string* function for several primitive types, defined within the *std* namespace. Whilst the C++ specification forbids the overloading of functions in the *std* namespace, the *to_string* function can be overloaded in the namespaces of user defined classes and the compiler will resolve them by examining the parameter type of the function call.

The Boost library also provides a *boost:lexical_cast<std::string>* function which relies on user defined *operator <<* functions to perform the to-string conversion.

The output of each of the above to-string implementations may differ.

In addition,

- calling the *to_string* function in a template class/function requires the systematic use of *using std::to_string* in order to resolve the built-in *to_string* overloads for the set of primitive types defined in the standard library *<string>* header,
- the definition of additional *to_string* function overloads for primitive types not included in the set of *to_string* overloads in the *<string>* header requires placing them outside of any namespace.

Unlike standard to-string functions or methods in other programming languages such as the *toString* method in Java, C++ does not have a unified standard for a to-string function, nor can it have a standard to-string method as there is no common base class to declare one in. Due to this and the complications described above, the Balau library standardises on the use of a single to-string function for each of the Unicode character encodings.

One possible solution to this requirement was to promote the primitive type *to_string* functions to the global namespace. This solution was decided against, in order to avoid using a token defined in the *std* namespace. In addition, three to-string functions (one per Unicode encoding) are required. Consequently, the *toString*, *toString16*, and *toString32* tokens were chosen instead.

Users of the Balau library may define *toString*, *toString16*, and *toString32* function overloads for their own custom types. Wrappers for the primitive type *std::to_string* functions defined in the standard library *<string>* header are also provided in the *<ToString.hpp>* header file. Additional overloads for common primitive types and standard containers are also supplied in *<ToString.hpp>*.

Signatures

The signatures of the universal UTF-8, UTF-16, and UTF-32 to-string functions are:

```
std::string toString(const T & value);
std::u16string toString16(const T & value);
std::u32string toString32(const T & value);
```

where *T* is the parameter type.

Usage

To use any of the Balau universal to-string functions, include the *<ToString.hpp>* header file in your code. As this header is already included in the *<BalauException.hpp>* header which is subsequently included in the *<Logger.hpp>* header, use of the logger or features that throw Balau exceptions will automatically include the *<ToString.hpp>* header file.

In order to provide universal to-string function overloads to Balau classes and functions for your custom types, it is sufficient to define a *toString*, *toString16*, or *toString32* function overload in the same namespace as your custom type. C++ argument-dependent lookup will resolve the function overload via the parameter type in the call.

Note that *toString*, *toString16*, or *toString32* function overloads should not be defined for type aliases, as this prevents the compiler from resolving the correct overload for a particular type. Instead, use the original type within its namespace.

When calling the *toString*, *toString16*, and *toString32* functions from a namespace that contains to-string function definitions in the namespace or a intermediate parent namespace, it may be necessary to import the the functions in the global namespace via a using directive, in order to ensure the correct overload is picked up from the local context.

```
// Example of using the toString function with a using directive.

struct G {};

std::string toString(G) {
    return "G";
}

namespace N {

class L {};

std::string toString(L) {
    return "L";
}

void foo() {
    using ::toString;

    std::cout << toString(L())     // Local scope.
              << toString(G())     // ADL.
              << toString(2)       // Requires using directive.
              << toString("hello") // Requires using directive.
              << "\n";
}

} // namespace N
```

Container to-string

The *ToString.hpp* header contains a template function *toStringHelper*. This helper function provides a convenient parameter pack template template parameter to-string implementation that can be selectively used for container types.

The declaration of the UTF-8 version of the helper function is as follows.

```
///
/// Helper for container to UTF-8 string functions.
///
/// This helper function can be used for custom container types if requirec
///
template <typename ... T, template <typename ...> class C>
inline std::string toStringHelper(const C<T ...> & c);
```

In order to use these container to-string helper functions, it is sufficient to define a new to-string function that calls the helper function. This can be done manually or via the *BALAU_CONTAINER**x**_TO_STRING**y*** macros, where ***x*** is the number of template parameters that the container accepts (1-5), and ***y*** is the unicode encoding (none for UTF-8, "16" for UTF-16, and "32" for UTF-32). These macros are also provided in the *ToString.hpp* header.

In order to use these macros, the container must implement ***begin() const*** and ***end() const*** iterator methods.

The *ToString.hpp* header contains a set of such functions for each of the standard library containers.

Parameter pack to-string

These versions of the to-string functions allow two or more input arguments to be converted to strings and concatenated together in a single function call. They are templated parameter pack versions of the universal to-string functions. They each contain a fold expression in order to concatenate string versions of the input arguments.

As with the other predefined universal to-string functions, there are UTF-8, UTF-16, and UTF-32 versions of the parameter pack to-string function.

The complete UTF-8 version of the parameter pack universal to-string function is as follows.

```
///
/// Calls toString on each input argument and concatenates them together to
/// form a single UTF-8 string.
///
template <typename P1, typename P2, typename ... P>
inline std::string toString(const P1 & p1, const P2 & p2, const P & ... p)
    return toString(p1) + toString(p2) + (std::string() + ... + toString(p
}
```

To-string template class

The *ToString.hpp* header also contains an additional template class based version of the universal to-string functions. This version consists of a class template declaration plus three specialisations, one for each character type. These classes are useful when the universal to-string function needs to be called from within a class or function template and when the string type is provided by a template argument.

The three specialisations are proxies to the previous sets of *toString*, *toString16*, and *toString32* function overloads.

The complete UTF-8 version of the templated universal to-string class is as follows.

```
///
/// Convert the supplied object to a std::string by calling toString.
///
template <> struct ToString<char> {
    template <typename T> std::string operator () (const T & object) const
        return toString(object);
    }
};
```

Custom allocation

In addition to the *std::basic_string<CharT>* based *to-string* functions, the *ToString.hpp* header file includes a parallel set of templated *to-string* functions that accept a custom allocator. The goal of these alternative functions is to provide suitable *to-string* function implementations for components that use a *std::basic_string<CharT, std::char_traits<CharT>, AllocatorT>* string type.

The notable usage of these templated *to-string* functions is in the logging system. When optionally enabled, the logging system uses a custom allocator that allocates on statically allocated thread local buffers. In this configuration, the logging system uses the *to-string* functions that accept an allocator template argument.

Universal from-string

#include <Balau/Type/FromString.hpp>

This section outlines a development approach and supporting code in the Balau library for a universal from-string function for each of the supported unicode encoding string types. The universal *from-string* functions provide a standard way to define string to object type conversions that can be used within library components. Similarly to the universal to-string functions, these functions are used throughout the Balau library, will propagate to application code, and application developers can define additional to-string function overloads for their own types.

Signatures

The signatures of the universal UTF-8, UTF-16, and UTF-32 from-string functions are:

```
void fromString(T & destination, std::string_view value);
void fromString16(T & destination, std::u16string_view value);
void fromString32(T & destination, std::u32string_view value);
```

where *T* is the parameter type.

The type *T* must be copy assignable, move assignable, have public mutable fields, or must provide suitable setter methods in order to be used in a *from-string* function overload. If this is not the case, the type is unsuitable to have a universal from-string function overload and a custom named from-string function (with a different signature) should be created instead.

Usage

To use any of the Balau universal from-string functions, include the *<FromString.hpp>* header file in your code.

In order to provide universal from-string function overloads to Balau classes and functions for your custom types, it is sufficient to define a *fromString*, *fromString16*, or *fromString32* function overload in the same namespace as your custom type. C++ argument-dependent lookup will resolve the function overload via the parameter type in the call.

When calling the *fromString*, *fromString16*, and *fromString32* functions from a namespace that contains from-string function definitions in the namespace or a intermediate parent namespace, it may be necessary to import the the functions in the global namespace via a using directive, in order to ensure the correct overload is picked up from the local context.

From-string template class

The *FromString.hpp* header also contains an additional template class based version of the universal from-string functions. This version consists of a class template declaration plus three specialisations, one for each character type. These classes are useful when the universal from-string function needs to be called from within a class or function template and when the string type is provided by a template argument.

The three specialisations are proxies to the previous sets of *fromString*, *fromString16*, and *fromString32* function overloads.

The complete UTF-8 version of the templated universal from-string class is as follows.

```
///
/// UTF-8 specialisation of FromString<T>.
///
/// Converts the supplied std::string to an object of type T by calling fro
///
template <> struct FromString<char> {
    ///
    /// @param destination the destination value that is set via assignment
    /// @param value the string input
    /// @throw ConversionException when the conversion fails
    ///
    template <typename T>
    void operator () (T & destination, const std::string & value) const {
        fromString(destination, value);
    }
};
```

Command line parser

Overview

A compact command line parser. The options of a constructed parser are specified via a fluent API. Options with and without values are supported. Options can be specified with abbreviated and full names. A final value option is also supported.

The command line parser supports two styles of command line switches. The first style is *switch - space - value* (SSV). In this style, the switches supplied on the command line start with a "-" for abbreviated switches and "--" for full switches. For arguments with values, the value is separated from the switch with whitespace.

The second supported style is *switch - equals - value* (SEV). In this style, the switches supplied on the command line do not have a prefix. For arguments with values, the value is separated from the switch with an equals character "=".

The command line parser can be configured to use one or the other style. Alternative, it can be default constructed whereapon it will detect the style from the first switch's leading character.

Quick start

#include <Balau/Application/CommandLine.hpp>

Style

Here are some examples of command lines.

```
#
# Command line that uses the SSV style and that has a final value.
#
# -k    = switch without value
# --max = switch with value
#  foo  = final value
#
./myApp1 -k --max 4 foo

#
# Command line that uses the SEV style and that has a final value.
#
./myApp2 k max=4 foo

#
# Command line that uses the SSV style and that does not have a final valu
#
./myApp3 -k --max 4

#
# Command line that uses the SEV style and that does not have a final valu
#
./myApp3 k max=4
```

Configuration

Here is an example of a configured parser. The switches specified in *withOption* calls do not take "-" or "--" prefixes, regardless of whether the parser is to be configured as SSV, SEV, or detected.

The command line parser is a template class. The single typename is the key type. Typically, this is an enum, the elements of which are the keys.

```
// Pre-defined option keys used in code.
enum Key {
    KEY1, KEY2, KEY3, HELP
};

auto commandLine = CommandLine<Key>()
    .withOption(KEY1, "k", "key-one", true, "The first key.")
    .withOption(KEY2, "m", "key-two", true, "The second key.")
    .withOption(KEY3, "3", false, "Specify in order to use third style.")
    .withHelpOption(HELP, "h", "help", "Displays this help message")
    .withFinalValue();
```

This builds a parser with two options that have values, one option that does not have a value, plus a (key-less) final value option.

If the application's command line does not have a final (switch-less) value, the *.withFinalValue()* call should be omitted.

Each option has a key (*KEY1*, *KEY2*, *KEY3* in the example above). These keys are used to query the parsed data later on.

The previous example will detect the style from the first command line argument parsed.

More often, one or the other style is chosen for an application. In order to create the same parser as in the previous example that is configured as SSV, the *CommandLineStyle::SwitchSpaceValue* argument should be passed into the constructor.

```
auto commandLine = CommandLine<Key>(CommandLineStyle::SwitchSpaceValue)
    .withOption(KEY1, "k", "key-one", true, "The first key.")
    .withOption(KEY2, "m", "key-two", true, "The second key.")
    .withOption(KEY3, "3", false, "Specify in order to use third style.")
    .withFinalValue();
```

Similarly, if the parser should be configured to parse the SEV style only, the *CommandLineStyle::SwitchEqualsValue* argument should be passed into the constructor.

```
auto commandLine = CommandLine<Key>(CommandLineStyle::SwitchEqualsValue)
    .withOption(KEY1, "k", "key-one", true, "The first key.")
    .withOption(KEY2, "m", "key-two", true, "The second key.")
    .withOption(KEY3, "3", false, "Specify in order to use third style.")
    .withFinalValue();
```

Retrieving data

To parse a command line, call the *parse* method.

```
commandLine.parse(argc, argv, true);
```

The first two arguments in the parse call are the standard argc/argv argument of the main function.

The third argument in the parse call indicates whether the first argument should be ignored (i.e. the first argument is the executable path).

To obtain options from the parser, call the option verification and extraction methods. In addition to string extraction, methods are available for extracting option data in a variety of primitive types. Refer to the command line parser API documentation for details.

Some example code that extracts the options from the previous command line parser is shown below.

```
auto o1 = commandLine.getOption(KEY1);
auto o2 = commandLine.getOption(KEY2);
auto o3 = commandLine.getOption(KEY3);
auto fv = commandLine.getFinalValue();
```

Help text

The command line parser has a method called *getHelpText* that generates a multi-line help text string for use in command line help text.

Resources

Overview

The resource classes provide a unified approach to specifying, sourcing and obtaining resource streams via URIs.

There are two groups of classes defined in the *Balau::Resource* namespace:

- URIs;
- resources.

URIs specify resources and provide functionality specific to each URI type. Resources are a convenient way of obtaining read and (for certain resource types) write streams on the resources specified by the URIs.

Quick start

#include <Balau/Resource/*.hpp>

URIs

URI classes each specify a different type of URI. All URI classes are derived from the common *Uri* base class. The currently available URI classes are:

- File;
- Http;
- Https;
- ZipFile;
- ZipEntry.

The *Http* and *Https* URI classes derive from the abstract *Url* class.

URI classes may be used as stack based objects or on the heap in standard pointer containers.

```
// Stack based URI objects.
File file { "text.txt" };
Http wiki { "http://wikipedia.org" };

// Heap based URI objects.
auto src1 = std::unique_ptr<URI>(new File("text.txt"));
auto src2 = std::unique_ptr<URI>(new Https("https://en.wikipedia.org/wiki/
```

Heap based URI objects allow different types of URI to provide abstract functionality, such as providing resources.

Resources

Each URI class is accompanied by two or four resource classes. The abstract resource classes are:

- ByteReadResource;
- ByteWriteResource;
- Utf8To32ReadResource;
- Utf32To8WriteResource.

The first pair of abstract resource classes provide byte based resource reading and writing.

The second pair of abstract resource classes provide UTF-8 to UTF-32 resource reading and UTF-32 to UTF-8 resource writing. These resource types allow reading from UTF-8 streams and consuming in UTF-32, and producing in UTF-32 and writing as UTF-8.

Resource objects must be obtained from URIs, either in heap based, abstract form contained within *std::unique_ptr* containers, or in stack based, concrete form directly from concrete URI types.

```
// An abstract URI.
std::unique_ptr<URI> uri = getUri();

// Get an abstract byte read resource from the URI.
std::unique_ptr<ByteReadResource> readResource = uri->byteReadResource();
```

Once a read / write resource has been obtained from a URI, the input / output stream (UTF-8 or UTF-32) may be obtained by calling *readStream* or *writeStream*.

```
// Get an input stream from the resource.
std::istream & stream = readResource->readStream();
```

URI classes

There are two ways to use the URI classes. The first way is to use instances of the URI classes directly as stack based objects. Balau components make extensive use of URI classes in this way, predominantly the *File* class. Each URI class provides an API that is specialised for URIs of the type provided by the class. For example, the *File* class provides the usual file manipulation functions such as checking for file existence and type, delete the file, etc.

The second way to use the URI classes is on the heap inside a pointer container. Some Balau components accept a *std::unique_ptr<Uri>* argument, allowing any type of URI to be supplied.

The most useful use cases for supplying *std::unique_ptr<Uri>* arguments to a component is:

- to obtain abstract read and/or write resources;
- to obtain an abstract recursive iterator.

The methods in the base *Uri* class that obtain read and write resources are as follows.

```
///
/// Get a byte read resource for the URI.
///
/// This will throw a NotImplementedException if the URI does not support

///
public: virtual std::unique_ptr<ByteReadResource> byteReadResource() = 0;

///
/// Get a UTF-8 to UTF-32 read resource for the URI.
///
/// This will throw a NotImplementedException if the URI does not support
///
public: virtual std::unique_ptr<Utf8To32ReadResource> utf8To32ReadResource

///
/// Get a byte write resource for the URI.
///
/// This will throw a NotImplementedException if the URI does not support
///
public: virtual std::unique_ptr<ByteWriteResource> byteWriteResource() = 0

///
/// Get a UTF-32 to UTF-8 write resource for the URI.
///
/// This will throw a NotImplementedException if the URI does not support
///
public: virtual std::unique_ptr<Utf32To8WriteResour
```

In order to determine whether a URI supports reading and writing, the *canReadFrom* and *canWriteTo* methods can be called.

```
///
/// Can data be read from the URI via a read resource.
///
public: virtual bool canReadFrom() const = 0;

///
/// Can data be written to the URI via a write resource.
///
public: virtual bool canWriteTo() const = 0;
```

The method in the base *Uri* class that obtains a recursive iterator is as follows.

```
///
/// Get a recursive iterator.
///
/// This will throw a NotImplementedException if the URI does not have a re
///
public: virtual std::unique_ptr<RecursiveUriIterator> recursiveIterator() 
```

In order to determine whether a URI supports recursive iteration, the *isRecursivelyIterable* method can be called.

```
///
/// Does the URI have a recursive iterator (e.g. file and zip archive URIs)
///
public: virtual bool isRecursivelyIterable() const = 0;
```

Resource classes

As discussed in the previous section, the abstract resource classes are:

- ByteReadResource;
- ByteWriteResource;
- Utf8To32ReadResource;
- Utf32To8WriteResource.

Byte based *ByteReadResource* and *ByteWriteResource* derived resources provide input and output streams that read/write as bytes. These can be used for any byte oriented purpose, including UTF-8 strings.

Unicode based *Utf8To32ReadResource* and *Utf32To8WriteResource* derived resources provide input and output streams that provide an internal format of *char32_t*, despite reading or writing the external resource data as a UTF-8 byte stream. These can be used when a stream of Unicode characters needs to be read or written.

Each concrete URI type provides at least one set of read or write resource classes. Some URI classes provide both sets of resource classes.

Recursive iterators

In addition to the resource readers and writers, some of the URI classes provide recursive iterators. These iterators supply a stream of new URIs, either of the same URI type (in the case of the *File* URI), or a different type (*ZipFile*). Resource readers/writers may be obtained from each of the URIs obtained during iteration.

The two types of recursive iterator currently defined in the library allow iteration over:

- the set of files and directories within a file system directory (*File*);
- the set of files and directories within a zip archive (*ZipFile*).

As the recursive iterators may be obtained from a pointer of type *Uri*, components may be written that consume input and/or output streams for sets of resources on the local file system or from within zip archives.

Custom resources

As the *Uri* and resource readers/writers are abstract classes, custom URIs and associated resources may be created easily and used within components that consume abstract resources. In order to do this, the *Uri* class should be implemented, along with associated implementations of *ByteReadResource* / *Utf8To32ReadResource* if the resource may be read, and *ByteWriteResource* / *Utf32To8WriteResource* if the resource may be written.

CONTAINERS

ArrayBlockingQueue

Overview

A simple blocking queue that uses a vector as the backing store.

ArrayBlockingQueue implements the *BlockingQueue* API and provides a blocking queue implemented via a wait-notify pattern.

Quick start

#include <Balau/Container/ArrayBlockingQueue.hpp>

To construct an array blocking queue, specify the capacity of the queue via the constructor.

```
// Create an array blocking queue with a capacity of 100.
ArrayBlockingQueue<T> queue(100);
```

The queue is used in the same way as any other *BlockingQueue* implementation. See the BlockingQueue API documentation for information on the blocking queue interface.

Concurrency

This queue implementation is thread safe but is not lock free.

DependencyGraph

Overview

A mutable graph structure that models dependency relationships in a dependency graph.

DependencyGraph uses the Boost graph library, and is inspired by the Boost graph library dependency example.

Quick start

#include <Balau/Container/DependencyGraph.hpp>

Construction

The construction of a dependency graph is made by the default constructor.

```
// Create a dependency graph (T is the value type stored in each graph node
DependencyGraph<T> graph;
```

Population

Population of the graph is performed with two main actions:

- adding dependency instances;
- creating dependency relationships.

These two actions correspond to the addition of graph nodes and graph edges.

To add a node, call the *addDependency* method.

```
// Add a dependency instance to the graph.

// The instance (normally sourced elsewhere).
T value;

graph.addDependency(value);
```

To add a dependency relationship, call the *addRelationship* method.

```
// Add a relationship between two dependencies.

// The instances (normally sourced elsewhere).
T independent;
T dependent;

graph.addRelationship(independent, dependent);
```

The dependency graph is designed to use small values that are used as keys. The types used in the dependency graph must therefore have a valid equals method for use in a map based structure. If a large amount of data needs to be stored in each value, one suitable approach would be to add a *std::shared_ptr* field to the value and exclude it from the comparison method logic.

Querying

In addition to standard iterators, the dependency graph has a set of query methods.

Function name	Description
hasDependency	Does the graph have the specified dependency?
directDependenciesOf	What are the direct dependencies of the specified dependency.
dependencyOrder	Calculate the dependency order of the dependencies.
parallelDependencyOrder	Calculate the parallel dependency order of the dependencies.
hasCycles	Does the dependency graph have any cycles?

In addition to the query methods, the *logGraph* method logs the contents of the dependency graph to the logging system.

See the DependencyGraph API documentation for information on the API for these methods.

Concurrency

The dependency graph is not thread safe.

ObjectTrie

Overview

An object based trie used for parent-child hierarchies.

This data structure is useful when the parent-child relationships between nodes are semantically important and must not be changed via tree balancing. If this is not the case, a B-tree or red–black tree would most likely be more appropriate.

Each node in the trie contains an object of type T plus a vector of child nodes. A node's children are ordered in the order of appending.

In addition to depth first and breadth first iteration, the object trie contains search and cascade algorithms. The object type T is thus effectively a combination of primary key and value. The type T normally provides a key type field that is compared within the *operator ==* function/method, and one or more value fields that are not part of the equality evaluation.

When using the search algorithms of the object trie, the standard operator == function / method is used for the object type T. Alternative methods are also provided that accept a custom compare function/lambda, allowing value types that do not have a suitable *operator ==* function/method to be used in the object trie search and cascade algorithms.

This trie implementation is not optimised or compressed. Use cases requiring an optimised trie representation would most likely be better using a non-object based trie implementation.

Quick start

#include <Balau/Container/ObjectTrie.hpp>

Construction

An object trie can be constructed either by constructing a trie with a root node containing a default constructed value, or by copying/moving a value into the trie to form the root node.

```
// Construct an object trie with a default root value.
// The int type is used as the value type.
ObjectTrie<int> trie1;

// Construct an object trie with a specified root value.
ObjectTrie<int> trie2(123);
```

When a class type is used as the object trie type, a suitable *operator ==* function/method should be defined if the object trie's algorithms are to be used.

```
// A class used in the object trie.
// The primary key of the class is the integer.
struct A {
    int k;
    double v;

    // Implicit construction for key only objects.
    A(int k_) : k(k_) {}

    // Explicit construction for full objects.
    A(int k_, double v_) : v(v_) {}
};

inline bool operator == (const A & lhs, const A & rhs) {
    return lhs.k == rhs.k;
}

ObjectTrie<A> trie({ 0, 123.45 });
```

The nodes of the trie are represented by *ObjectTrieNode* objects. These contain the value T, and a vector of child nodes.

Trie nodes

The root node of the trie may be obtained via the *root* method.

```
ObjectTrieNode<A> & rootNode = trie.root();
```

To obtain child nodes, the trie has *count* and *get* methods that provide the number of children of the root and references to the child nodes. Similarly, the *ObjectTrieNode* has equivalent *count* and *get* methods that provide the number of children of the node and references to the children of the node.

```
// Get the second child of the root node.
ObjectTrieNode<A> & c = trie.get(1);

// Get the first child of the second child.
ObjectTrieNode<A> & c = c.get(0);
```

New child nodes of the root node or child nodes may be added via the *add* and *addAndReturnChild* methods. The *add* method adds the new child node and returns the current node. The *addAndReturnChild* method adds the new child node and returns the new child.

```
// Add a new node to the root of the trie.
trie.add(2);

// Add a new node to the first child of the root node.
c.get(0).add(7);
```

Searching

The *find* and *findNearest* methods perform hierarchical exact and nearest searching.

Find

To use the *find* methods, call one of the methods with a vector of values to compare. The find method will descend into the trie, comparing each supplied value in turn with the current set of children. If a match is not found in one of levels, *nullptr* is returned.

There are two *find* method overloads. The first overload uses the default *operator ==* function /method for the type *T* and the second overload allows a custom comparator to be specified.

```
// Perform an exact search with the default comparator.
auto * n = trie.find({ 1 });
```

By default, the root node is not included in the search, thus the first object in the supplied vector is compared with the child nodes of the root node.

If the root node should be included in the search, *true* should be passed as the second argument in the method call.

```
// Perform an exact search with the default comparator.
// Include the root node in the search.
auto * n = trie.find({ 0, 1 }, true);
```

In order to use a custom comparator, a lambda function can be specified in the call.

```
// Perform an exact search with a custom comparator.
auto * n = trie.find(
      { 0, 1 }
    , [] (auto & lhs, auto & rhs) { return lhs.k == rhs.k; }
);
```

FindNearest

The *findNearest* methods work in the same way as the *find* methods, with the exception that if a match is not found in one of levels, the current match is returned. If no matches are found at all, then *nullptr* is returned.

```
// Perform a nearest search with the default comparator.
auto * n = trie.findNearest({ 1, 2, 3 });
```

FindNearestLeaf

The *findNearestLeaf* methods work in the same way as the *findNearest* methods, with the exception that a match must terminate with a leaf node. If a nearest match is found that is not a leaf node, then *nullptr* is returned.

```
// Perform a nearest leaf search with the default comparator.
auto * n = trie.findNearestLeaf({ 1, 2, 3 });
```

Iteration

Standard library compatible iterators are provided in the object trie implementation. There are two types of iterator available:

- depth first;
- breadth first.

The *iterator* and *const_iterator* iterators are typedefs to the *DepthIterator* and *ConstDepthIterator* object trie iterators.

Depth first

Depth first iteration is performed in the same way as iteration in any standard library container.

```
ObjectTrie<int> trie(0);
populateUIntTrie(trie);

// Traditional iteration.
while (ObjectTrie<int>::iterator i = trie.begin(); i != trie.end(); ++i) {
    auto & object = i->value;

    // ... use the object ...
}

// Range-based iteration.
for (auto & node : trie) {
    auto & object = node.value;

    // ... use the object ...
}
```

If the depth first iteration should be explicitly mentioned in code, the *depthBegin* and *depthEnd* calls can be specified instead of *begin* and *end*.

```
while (auto i = trie.depthBegin(); i != trie.depthEnd(); ++i) {
    auto & object = i->value;

    // ... use the object ...
}
```

Breadth first

As the range-based loop defaults to depth first iteration, breadth first iteration must be performed via traditional iteration.

```
// Perform a breadth first iteration on the same trie.
while (auto i = trie.breadthBegin(); i != trie.breadthEnd(); ++i) {
    auto & object = i->value;

    // ... use the object ...
}
```

Cascading

Object trie cascading involves copying or moving one object trie onto another object trie with the following rules.

- Each node in the source trie that matches a node in the destination trie for equality and position is (copy or move) assigned to the node in the source trie.
- Each node in the source trie that does not match a node in the destination trie for equality and position is (copy or move) added to the destination trie.

```
// Perform an object trie cascade.
ObjectTrie<int> trie1(0);
populateUIntTrie1(trie1);

ObjectTrie<int> trie2(0);
populateUIntTrie2(trie2);

trie1.cascade(trie2);
```

The above code performs copy cascading. This results in the source trie maintaining validity after the cascade operation. If move cascading is required instead, the *std::move* cast may be used to move the source trie into the cascade call.

```
// Perform an object trie cascade via moving.
trie1.cascade(std::move(trie2));
```

The third and fourth *cascade* methods overloads accept a copy or move function in addition to the source trie. This allows the source node's value to be modified during copying / moving. Otherwise, the cascade semantics are identical to the first two method overloads.

Fluent build API

In order to provide a visual representation of an object trie in code, a variadic fluent build API is provided in the object trie implementation. This build API can be most useful in test code, where canned tries need to be constructed.

The *ObjectTrieTest::fluentBuild* test case provides a usage example of the fluent build API.

```
using Trie = ObjectTrie<Value>;
using Node = ObjectTrieNode<Value>;

Trie trie;

trie.add({ 'a', 1 }
    , Node::child({ 'a', 11 })
    , Node::child({ 'b', 12 })
    , Node::child({ 'c', 13 })
    , Node::child({ 'd', 14 })
).add({ 'b', 2 }
    , Node::child({ 'a', 21 })
    , Node::child({ 'b', 22 }
        , Node::child({ 'a', 221 })
        , Node::child({ 'b', 222 })
        , Node::child({ 'c', 223 })
    )
).add({ 'c', 3 }
    , Node::child({ 'a', 31 })
    , Node::child({ 'b', 32 })
    , Node::child({ 'c', 33 })
    , Node::child({ 'd', 34 })
    , Node::child({ 'e', 35 })
);
```

SharedMemoryQueue

Overview

A blocking, shared memory queue that uses the Boost Interprocess library.

This class provides a shared memory backed blocking queue and implements the *BlockingQueue* API. The class encapsulates calls to the Boost Interprocess library, which create, use, and delete a shared memory queue, and has additional sequencing and chunking management logic. The sequencing and chunking logic manages the enqueueing and dequeueing of oversize objects, which require multiple shared memory queue send and receive operations that may be out of order and/or interleaved with other enqueued buffers.

The implementation uses the Boost Serialization library for marshalling and unmarshalling of objects. In order to use the queue, the object type *T* must provide Boost *serialize* or *save/ load* methods.

Quick start

#include <Balau/Interprocess/SharedMemoryQueue.hpp>

The queue is used in the same way as any other *BlockingQueue* implementation.

The queue can be instantiated in three ways:

- as the creator of the shared memory objects in the queue;
- as the creator or user of the shared memory objects in the queue;
- as a user of the shared memory objects in the queue.

Create

The most simple constructor used to create a queue is as follows.

```
// Create a shared memory queue for objects of type T and
// with a capacity of 100.
SharedMemoryQueue<T> queue(100);
```

Such a queue is only useful if the application will share the queue by forking or if the automatically generated name prefix is obtained by calling *getName* on the resulting queue instance.

In order to create a queue with a known name prefix, use the constructor that takes a string argument in addition to the capacity.

```
// Get the queue's predefined name prefix from somewhere.
const std::string name = getQueueName();

// Create a shared memory queue with the predefined name prefix.
SharedMemoryQueue<T> queue(100, name);
```

SharedMemoryQueue has a number of other optional parameters. These are outlined below.

Parameter	**Type**	**Default**	**Description**
capacity	unsigned int	No default	The number of items that the queue can hold.
buffer size	unsigned int	Marshal size of T() plus header	The size in bytes of each item in the queue. This size includes the header size.
name	std::string	UUID	The name of the queue.
throw on oversize	bool	false	Throw an exception if an attempt to enqueue an oversize object is made.

If the queue is to be used with multiple dequeueing processes, the buffer size of the queue must be large enough to fit all serialised objects plus the queue header size of 16 bytes. For POD objects, which do not have any fields that allocate memory, the default buffer size calculated by the queue from a default constructed object is sufficient. For object types that do have fields that allocate memory, the default buffer size calculated by the queue is not sufficient and thus the buffer size must be supplied manually. Otherwise, the queue will be defective.

Note that manual specification of the buffer size is not required if dequeueing will occur in a single process and with synchronised access. In this case, the result of not specifying a sufficient buffer size will result in oversize serialisations being split into chunks which are then sent over the shared memory queue in turn. Then, the single process calling *dequeue/ tryDequeue* with synchronised access will join these chunks together before deserialising.

Open or create

An equivalent pair of constructors are available that open the shared memory objects of a queue if they already exist, otherwise, they create the objects. In order to use these constructors, the *OpenOrCreateSelector* object must be passed as the first argument. Otherwise, these constructors are identical to their counterparts which only create the shared memory objects.

```
// Get the queue's predefined name prefix from somewhere.
const std::string name = getQueueName();

// Open or create a shared memory queue with the predefined name prefix.
SharedMemoryQueue<T> queue(OpenOrCreateSelector, 100, name);
```

Open

There is a single constructor for instantiating a *SharedMemoryQueue* as a user of an existing queue. This constructor takes a *std::string* containing the name of the queue to open.

```
// Get the queue's predefined name prefix from somewhere.
const std::string name = getQueueName();

// Open a shared memory queue with the predefined name prefix.
SharedMemoryQueue<T> queue(name);
```

Usage

Once the queue has been created or opened, it can be used in the same way as any other *BlockingQueue* implementation. See the BlockingQueue API documentation for information on the blocking queue interface.

Concurrency

This queue implementation has the following concurrency guarantees.

The queue can be used for concurrent enqueues and concurrent dequeues across processes/threads if the maximum enqueued serialised object size + queue header size is guaranteed to be smaller than the shared memory queue buffer size.

If the above guarantee cannot be met (due, for example, a non-deterministic serialised object size), the queue can be used for concurrent enqueues across processes/threads, but only synchronised dequeues in a single process. This is due to the dequeueing of partial objects occurring in one process, rendering the continuation of the dequeueing of that object impossible in other processes.

If this limitation is breached, the set of applications using the shared memory queue will be defective.

The dequeueing calls in such a scenario must also be protected by a mutex if multiple threads of the dequeueing application are concurrently dequeueing. No such mutex protection is required if oversize objects are not being enqueued.

In order to catch oversize message errors in a system that is not designed for oversize message dequeueing, all constructors of the *SharedMemoryQueue* accept an additional boolean argument. Setting this argument to true will cause an exception to be thrown if an attempt is made to enqueue an oversize message. This check can be switched on in order to catch early such errors during the development and testing phases.

Use cases

There are two ways to utilise a shared memory queue in multiple processes:

- by forking a parent process;
- by communicating the queue name to other processes.

Forked processes

Forking is a simple way to construct and use the share memory queue across processes, but it is only supported by Unix-like operating systems. In order to construct and use a shared memory queue in a parent process and a set of forked child processes, construct the queue in the parent and fork as normal. The Balau Fork class provides a convenient API for forking. The shared memory queue will be ready for use in the child processes without any further action. The first constructor is used for this.

```
// The type of object being sent across the queue.
struct A {
    int i;
    double d;

    A() : i(0), d(0.0) {}
    A(int i_, double d_) : i(i_), d(d_) {}

    // The serialize method, used by the queue to marshal and unmarshal the
    template <typename Archive> void serialize(Archive & archive, unsigned
        archive & BoostSerialization(d) & BoostSerialization(i);
    }
};

// Construct the shared memory queue before forking.
SharedMemoryQueue<A> queue(100);

// Perform the fork. The child will not return.
Fork::performFork([&queue] () { return runChildLogic(queue); }, true)
```

Independent processes

Processes that are not related by forking may access the same shared memory queue by communicating the name to each process.

There are two possibilities for communicating the name:

- pre-share the name between the processes;
- create the queue in one process and communicate its name to the other processes in some way.

With the first solution, a name is decided upon in advance or is algorithmically generated by the application. One solution to this when sharing a queue between multiple instances of the same application is to construct a name prefix via the application's executable path. A helper function namePrefixFromAppPath() is available for this in the *SharedMemoryUtils* class. Using this solution, a set of shared memory queue names can be created by appending predefined strings to the name generated from the helper function.

Another solution is to pre-share a name that can be guaranteed not to be used by other processes, either hard wired in the application (not recommended) or via the application's configuration/options.

In order to use a peer-to-peer approach, the *create-or-open* constructors can be used.

```
// Create the name for the shared memory queue.
const std::string name = SharedMemoryUtils::namePrefixFromAppPath() + "_my

// Create or open the shared memory queue with the name prefix.
SharedMemoryQueue<A> object(OpenOrCreate, 100, name);
```

In order to use a manager-worker approach, one of the constructors which creates the shared memory objects can be used in the manager process and the queue open constructor can be used in the worker processes. Due to the necessity of the queue existing for the worker processes, the manager process will need to create the queue before the workers attempt to open it.

```
// Manager process..

// Create the name prefix for the shared memory queue.
const std::string name = SharedMemoryUtils::namePrefixFromAppPath() + "_my

// Create the shared memory queue.
SharedMemoryQueue<A> queue(100, name);

/////////////////////////////////////////////////////////////////////

// Worker process..

// Create the name prefix for the shared memory queue.
const std::string name = SharedMemoryUtils::namePrefixFromAppPath() + "_my

// Open the shared memory queue.
SharedMemoryQueue<A> queue(name);
```

SynchronizedQueue

Overview

A simple non-blocking queue that uses a std::list and a mutex to provide a thread safe queue.

SynchronizedQueue implements the *Queue* API and provides a simple (and inefficient) thread safe queue.

Quick start

#include <Balau/Container/SynchronizedQueue.hpp>

The construction of a synchronised queue is made by the default constructor.

```
// Create a synchronised queue.
SynchronizedQueue<T> queue;
```

The queue is used in the same way as any other *Queue* implementation. See the Queue API documentation for information on the queue interface.

Concurrency

This queue implementation is thread safe but is not lock free.

CONCURRENT

CyclicBarrier

Overview

A synchronising barrier that can be configured for an arbitrary number of threads.

The barrier automatically resets after releasing the threads.

The barrier can be reconfigured for a different number of threads. It is the responsibility of the using code to ensure that the barrier is not being used when reconfigured.

Quick start

#include <Balau/Concurrent/CyclicBarrier.hpp>

Using the cyclic barrier is simple. The barrier is constructed or reconfigured with the number of threads that will wait before the barrier will release the threads and reset.

```
// Create a cyclic barrier.
CyclicBarrier barrier(4);

// Reconfigure the cyclic barrier.
barrier.reconfigure(2);
```

Then each participating thread calls the barrier.

```
/// Count down the barrier, blocking if the count has not reached 0.
barrier.countdown();
```

Fork

Overview

A wrapper around the *fork* and *waitid* functions.

The wrapper has two fork related functions and a set of functions used to manage child process termination.

The fork related functions are:

- a function to determine whether forking is supported for the platform;
- a function to perform the fork and run the supplied function in the child.

The child process termination management functions are:

- a function to wait on a child process;
- a set of functions to check for a terminated child process.

Quick start

#include <Balau/Concurrent/Fork.hpp>

Forking

The *forkSupported* function can be used to verify that the platform has fork support.

The *performFork* function overloads take a function object (pointer, lambda) which will be run in the child process after forking. There are two types of overload.

The first overload requires a function that returns an exit status. This exit status in the child process will be returned from the function if the *exitChild* boolean is set to false. Otherwise, the child process will exit at the end of the call to *performFork* with the exit status given by the supplied function. The parent process will return the PID of the child process from the *performFork* function.

```
/// /// Create a set of worker processes. /// ///

// The worker state to be shared across processes.
class WorkerState {
    // ... worker state fields ... //
}

using WorkerStateObject = MSharedMemoryObject<WorkerState>;
using WorkerStatePtr = std::shared_ptr<WorkerStateObject>;

// The worker implementation.
class Worker {
    private: WorkerStatePtr state;

    public: Worker(WorkerStatePtr state_) : state(std::move(state_)) {}

    public: int run() {

        // ... worker loop ... //

        return 0;
    }
};

// Create the shared state and workers.
WorkerStatePtr createWorkers() {
    // Create the shared state.
    auto workerState = std::make_shared<WorkerStateObject>();

    // Create the worker.
    Worker worker(sharedShared);

    std::vector<int> childProcessPids;

    // Create the child processes and run the workers within.
    for (unsigned int m = 0; m < WorkerCount; m++) {
        childProcessPids.push_back(
            Fork::performFork([&worker] () { return worker.run(); }, true)
        );
    }

    // ... Parent process continues with access to shared state ... //
    return workerState
}
```

The second overload in the *Fork* class does not require a function/lambda that returns any particular type. This overload will return 0 on successful completion of the supplied function in the child process and will return zero. The parent process will again return the PID of the child process from the *performFork* function. It is then the responsibility of the caller to handle the continuing execution of the child process.

Termination

There are three functions available for managing child process termination.

Function name	Description
waitOnProcess	Wait on a process until the process terminates.
checkForTermination	Check the process or processes for termination without blocking.
terminateProcess	Terminate the child process if it is running.

See the Fork API documentation for more information.

Semaphore

Overview

A traditional semaphore synchronisation object.

The semaphore maintains a set of permits that are created via *increment* calls on the semaphore, and consumed by *decrement* calls.

If there is no permit available when a *decrement* call is made, the calling thread blocks until a permit is created via an *increment* call.

Quick start

#include <Balau/Concurrent/Semaphore.hpp>

Using the semaphore is simple.

```
// Create a semaphore.
Semaphore semaphore;
```

An initial permit count can be specified at construction time if desired.

```
// Create a semaphore with an initial permit count of 4.
Semaphore semaphore(4);
```

Then each participating thread may call *increment* and/or *decrement* accordingly.

```
///// Thread A /////

/// Decrement the semaphore, blocking if the count is 0.
semaphore.decrement();
```

```
///// Thread B /////

/// Increment the semaphore, adding a permit.
semaphore.increment();
```

SharedMemoryObject

Overview

Shared memory objects that use the Boost Interprocess library.

This documentation covers two classes which are similar in usage:

- MSharedMemoryObject;
- USharedMemoryObject.

The semantic difference between these two classes is that instances of *USharedMemoryObject* require a call to *remap()* in child processes after forking, whilst instances of *MSharedMemoryObject* do not. The other difference between the two classes is that *MSharedMemoryObject* uses half a kilobyte of shared memory for metadata, whilst instances of *USharedMemoryObject* do not.

As operating systems generally allocate entire pages for shared memory (a page is typically 4KB), *MSharedMemoryObject* instances generally use the same amount of memory as *USharedMemoryObject*, unless your object has a size of between 4096N + 3584 and 4096 (N+1), where N is an unsigned integer.

Note also that as shared memory is typically allocated in 4k pages, a whole page will be allocated even if your object size is a single byte. It is thus not efficient to create a large number of managed/unmanaged shared memory object instances each containing a small object. If a large number of small objects need to be shared across multiple processes, the efficient approach is to create a holder class of these objects and then to create a single managed/unmanaged shared memory object of the holder.

These template classes encapsulate calls to the Boost Interprocess library in order to manage the lifetime of a shared memory object, and provide a simple API to construct/open and use the object.

The shared memory object classes are useful when a simple approach to creating a typed shared memory object is desired. They can also act as a tutorial introduction into using shared memory via the Boost Interprocess library. More advance use of shared memory can then follow by direct use of the Boost library.

Note that when using the shared memory object classes, the type T must have a POD type structure. If the type contains pointers, the objects pointed to will not share and consequently your application will be defective. If non-POD data structures are required to be shared across multiple processes, advanced use of the Boost Interprocess library is recommended.

Quick start

#include <Balau/Interprocess/MSharedMemoryObject.hpp>
#include <Balau/Interprocess/USharedMemoryObject.hpp>

There are two types of constructors in these classes:

- one constructor that creates automatically named shared memory objects;
- constructors that explicitly create or open a new or existing named shared memory object.

The first constructor creates an automatically named shared memory object by generating a name prefix based on a UUID.

The second set of constructors take a dummy object specifying whether to create or open the shared memory object, plus the name of the shared memory.

Forked processes

Forking is a simple way to construct and use the share memory object across processes, but it is only supported by Unix-like operating systems. In order to construct and use a shared memory object in a parent process and a set of forked child processes, construct the object in the parent and fork as normal. The Balau Fork class provides a convenient API for forking. The shared memory object will be ready for use in the child processes without any further action. The first constructor is used for this.

```
// The type of object being shared.
struct A {
    int i;
    double d;

    A(int i_, double d_) : i(i_), d(d_) {}
};

// Construct the shared memory object before forking.
MSharedMemoryObject<A> sharedA(1, 2.0);

// Perform the fork. The child will not return.
Fork::performFork([&sharedA] () { return runChildLogic(sharedA); }, true)
```

As previously discussed, use of the *USharedMemoryObject* class with forking will require subsequent calls to *remap()* in each child process before the object is usable. The equivalent code to the previous example using an instance of *USharedMemoryObject* is thus as follows.

```
// Construct the shared memory object before forking.
USharedMemoryObject<A> sharedA(1, 2.0);

// Perform the fork. The child will not return.
Fork::performFork(
    [&sharedA] () {
        sharedA.remap();
        return runChildLogic(sharedA);
    }
    , true
)
```

Independent processes

Processes that are not related by forking may access the same shared memory object by communicating the name prefix to each process. A choice of constructors are available for this, which implement the Boost Interprocess *create-only*, *create-or-open*, *open-only*, and *open-read-only* options.

There are two possibilities for communicating the name prefix:

- pre-share the name prefix between the processes;
- create the object in one process and communicate its name to the other processes in some way.

With the first solution, a name prefix is decided upon in advance or is algorithmically generated by the application. One solution to this when sharing an object between multiple instances of the same application is to construct a name prefix via the application's executable path. A helper function namePrefixFromAppPath() is available for this in the *SharedMemoryUtils* class. Using this solution, a set of shared memory object name prefixes can be created by appending predefined strings to the name prefix generated from the helper function.

Another solution is to pre-share a name prefix that can be guaranteed not to be used by other processes, either hard wired in the application (not recommended) or via the application's configuration/options.

The template constructors are used for creating/opening a shared memory object with a pre-shared or algorithmically generated name.

In order to use a peer-to-peer approach, the *create-or-open* constructor can be used.

```
// Create the name prefix for the shared memory object.
const std::string name = SharedMemoryUtils::namePrefixFromAppPath() + "_my

// Create or open the shared memory object with the name prefix.
USharedMemoryObject<A> object(OpenOrCreate, name, 2, 4.0);
```

In order to use a manager-worker approach, the *create-only* constructor can be used in the manager process and the *open-only* constructor can be used in the worker processes. Due to the necessity of the queue existing for the worker processes, the manager process will need to create the queue before the workers attempt to open it.

```
// Manager process..

// Create the name prefix for the shared memory object.
const std::string name = SharedMemoryUtils::namePrefixFromAppPath() + "_my

// Create the shared memory object.
USharedMemoryObject<A> object(CreateOnly, name, 2, 4.0);

///////////////////////////////////////////////////////////////////

// Worker process..

// Create a name prefix for the shared memory object.
const std::string name = SharedMemoryUtils::namePrefixFromAppPath() + "_my

// Open the shared memory object.
USharedMemoryObject<A> object(OpenOnly, name);
```

LANG

Parsing utilities

Overview

Balau includes a set of classes which can help in the construction of language scanners and parsers. The aim of these classes is to facilitate the creation of hand written language scanners and recursive descent parsers written in pure C++.

Given that there are many mature scanner/parser generator tools available, one pertinent question to ask may be: why write a scanner and parser in pure C++?

Firstly, there is no right or wrong approach to writing a scanner/parser pair. If a generator tool works well for a particular use case, then such an approach is admirable. However, given that the parsers of prominent mainstream compilers are hand written (Clang - unified parser, GCC - new C parser), perhaps there are valid reasons for doing so.

One anecdotal reason often quoted is performance, that is to say that a hand written parser will supposedly be much faster than a generated one. The evidence does not appear to back this up (the GCC wiki indicates a negligible 1.5% speed increase). There may thus be a small percentage speed up, but it is unlikely that a speed up measured in orders of magnitude will occur.

So removing performance from the argument, some benefits are proposed below.

- **Fine tuning of semantics** - Language definitions of any complexity will often require additional semantic rules that are expressed separately from the language's grammar specification. These additional semantic rules may not fit naturally into the language definition processed by a parser generator tool.
- **Debuggability** - The code is hand written pure C++ and thus can be structured for readability and easy debugging.
- **Error reporting, diagnostics, recovery** - Precise error reporting, diagnostics, and recovery are easier to achieve when the complete and final scanner/parser source code is available to edit directly.
- **Elegance** - A hand written recursive descent parser is an elegant solution to a complex requirement.
- **Simple toolchain** - No additional tools are required for parser source code generation.

The overriding reason to use scanner/parser generator tools is that the amount of code you need to write is much less than the code required for a hand written scanner and parser. This is most likely true. However, a language specification is typically created once and then subsequently modified over time with only small incremental improvements. The overhead of creating the larger code base of a hand written scanner and parser is thus likely to be less than the benefits reaped during fine tuning of the initial implementation and the ease of adding incremental improvements later on.

Approach

Having read the introductory words in the overview section of this page, it could be natural to imagine that the parsing utilities in the Balau library are large and complex. This is actually not the case. The parsing utilities total less than 1000 lines of code and a handful of classes.

As the aim is to hand write a scanner and parser implementation pair, the value of the utilities lies in the approach rather than the amount of code provided. The parser tools do not actually provide any classes for parsing at all. The code only provides an abstract scanner base class and a common contract between the scanner and the to be written parser.

The overall approach to the creation of a new pure C++ based language parser using the Balau parsing utilities classes is to:

- define a token enum that contains the terminals of the language;
- create a scanner class that derives from *AbstractScanner* and produces a *ScannedTokens<TokenT>* data structure;
- create a set of abstract syntax tree node classes which represent the non-terminals of the language;
- create a recursive descent parser class which generates the AST by consuming the *ScannedTokens<TokenT>* data structure via the *ScannerApiScannedTokens* token adaptor.

There are currently two language parser implementations in the Balau library that follow this approach:

- the logging configuration parser in the logging framework;
- the hierarchical property parser.

Before embarking on the creation of a language parser based on the Balau parser utility classes, it could be useful to take a look at these parser implementations.

Architecture

The overall architecture supported by the Balau parser utilities is one where:

- there is complete separation between scanning and parsing stages;
- the scanning stage scans the entire text stream ahead of parsing;
- the resulting scanned tokens data structure offers infinite look-ahead and look-back;
- the whitespace policy during parsing may be dynamically modified via the whitespace mode stack in the scanner API scanned tokens data structure.

The advantages of such an approach are the separation of concerns between the scanning and parsing stages, the infinite look-ahead and look-back, and the ease of dynamically changing the whitespace handling strategy during parsing.

The disadvantage of such an approach is the necessity of specifying a single token set which covers all possible tokens of the language, i.e. it is not possible to define multiple token sets, the active set then being selected according to context during the parsing stage.

There are a number of ways of mitigating this disadvantage when a parser cannot be trivially implemented with a complete separation between scanner and parser for a particular language grammar. One solution could be to define a single token for the contextual part of the language and then define a second scanner which is triggered separately from within the parser. Another solution could be to define a set of tokens that represent the input text and then, under certain exceptional code paths in the parser, translate subsets of the input tokens on the fly into a modified set of tokens for further parsing.

Scanned tokens

#include <Balau/Lang/Common/ScannedTokens.hpp>

An instance of the *ScannedTokens<TokenT>* class is generated by the scanner and consumed by the parser. This class is an efficient data structure which contains the input text as a UTF-8 string and two additional arrays:

- an array that contains the tokens, which are typically specified as an 8 bit unsigned char typed enum;
- a second array that contains the start offsets into the input text.

In order to use the scanned tokens data, instantiation of an adaptor class is required. There are three adaptor classes, destined for different functionalities:

- ScannerApiScannedTokens;
- RandomAccessScannedTokens;
- IterativeScannedTokens.

Scanner Api

With the *ScannerApiScannedTokens* adaptor class, a standard scanner API is provided for use by a traditional downstream parsing stage. This is the adaptor that is used if traditional parsing is required.

The *ScannerApiScannedTokens* adaptor class provides the following methods.

Method	Description
get	Get the current scanned token, optionally consuming blanks, line breaks, and/or comments as determined by the current whitespace mode.
consume	Consume the current token.
expect(token)	Consume the current token if it is equal to the supplied token, otherwise register an error report or throw an exception.
expect(tokens)	Consume the current token if it is equal one of the supplied tokens, otherwise register an error report or throw an exception
putBack	Put back the current token, optionally restoring blanks, line breaks, and /or comments as determined by the current whitespace mode.
mark	Record a token position marker for a subsequent multiple putBack call.
putBack(marker)	Put back to the specified marker.
ushWhitespaceMode	Push the specified whitespace mode onto the whitespace mode stack.
opWhitespaceMode	Pop the top of the whitespace mode stack.
etCurrentCodeSpan	Get the code span for the current token's text.
reset	Reset the scanner state to the beginning of the token list and with an empty whitespace mode stack.
size	Get the number of tokens produced from the scanned text.
moveTextOut	Move the text of the scanner out (prevents further scanning calls).

The *expect* methods that register an error report do so via a supplied error report function that produces an error report. The report is added to the supplied container. These *expect* methods should be used if an error recovery type parser is required, as opposed to a fail on first error parser.

Due to the compact representation of the scanned token data, obtaining the code span of a token's text must be performed by iterating from the start of the arrays. When using the *ScannerApiScannedTokens* adaptor class, the current code span is maintained for immediate use. Due to this, the current code span may be accessed without any performance penalty.

The scanner API also provides classification methods for the current token and token specified by a marker:

- currentIsBlank;
- currentIsLineBreak;
- currentIsWhitespace;
- isBlank(marker);
- isLineBreak(marker);
- isWhitespace(marker);

Random access

If random access to the scanned token data and code spans is required for other uses, the *RandomAccessScannedTokens* adaptor class may be used to pre-calculate the code spans. When instantiated, this class iterates over the token set and calculates all the code spans. Due to this, the *RandomAccessScannedTokens* class has a higher memory overhead than the *ScannerApiScannedTokens* class.

Iteration

When iteration is required over the scanned tokens, this third adaptor class is provided for such applications. The *IterativeScannedTokens* adaptor class provides standard C++ iterators onto the scanned tokens.

Scanning

#include <Balau/Lang/Common/AbstractScanner.hpp>

In order to create a *ScannedTokens* data structure from some input source code text, a scanner implementation is required. Balau provides an *AbstractScanner* base class. Concrete implementations of this class implement the *getNextToken* abstract method in order to provide the scanning logic.

The *AbstractScanner* class provides the public scanner api, consisting of the single *scan* method, and a set of protected utility methods used by implementing classes.

The fundamental utility methods used by implementing classes are the *readNextChar* and *putBackCurrentChar* methods. These manage character reading and end of file handling.

Other utility methods including string and whitespace extraction. Also included is a *calculateCurrentCodeSpan* method which can be used for error reporting.

Parsing

To create a recursive descent parser in pure C++ using the Balau parser utility classes, it is sufficient to pass in an instance of the *ScannerApiScannedTokens* class into the parser constructor and access the scanner API from within the production methods.

The *PropertyParser* class is an example of a parser in the Balau library. A single public method *parse* is provided in each class. This method calls the root production methods in order to initiate the recursive descent parsing.

Classes

The following classes are defined in the parsing utilities.

Class name	Description
ScannedTokens<TokenT>	The data structure produced by scanners. Contains the input source text, the scanned tokens, and the start offsets.
ScannerApiScannedTokens<TokenT>	A scanner API adaptor over the *ScannedTokens* data structure.
ndomAccessScannedTokens<TokenT>	A random access adaptor over the *ScannedTokens* data structure.
IterativeScannedTokens<TokenT>	An iterative adaptor over the *ScannedTokens* data structure.
AbstractScanner<TokenT>	The base class of scanner implementations.
ScannedToken<TokenT>	A single scanned token returned from the scanner API, containing the token, the text, and the code span.
CodeSpan	Contains two code positions which together indicate a span of code.
CodePosition	Represents a line/column position in the source code text.

Hierarchical properties

Overview

This chapter describes a hierarchical property file format and associated C++ parser. The format has been conceived principally for describing application, environment, and logging configuration. The format is based upon a hierarchical extension to the Java *.properties* file format. Composite properties are defined via *"{"* and *"}"* delimited blocks.

In addition to providing composite properties, the hierarchical property format provides an *include* feature. This allows property files to be spread over multiple files and compiled via inclusion. Includes are specified via the *"@"* directive.

The hierarchical properties parser and scanner are written in pure C++, using the Balau parser utility classes.

The hierarchical properties parser can also parse non-hierarchical Java properties files. The exception that cannot be parsed is when a non-hierarchical properties file contains one or more names and/or values that contain unescaped special hierarchical characters. As the *"{"* , *"}"* and *"@"* characters are used to indicate hierarchical blocks and include directives, parsing will fail if names and/or values are defined with these characters in them. Property names and values that contain these character can nevertheless be defined by escaping the special characters with a *"\"*.

No file extension has been explicitly denoted to indicate hierarchical properties files. Given that the hierarchical property file format is effectively just a text based serialisation format, files themselves do not have any intrinsic semantics. It is thus proposed that users should define their own file extensions which attach semantic value to file contents. The Balau library thus uses the *.properties* extension for hierarchical properties files without specific semantics, the *.thconf* extension for environment configuration type specification files, and the *.hconf* extension for environment configuration value files. The environment configuration chapter discusses this in more detail.

Quick start

Format

The following is a simple example of a hierarchical property file. In the example file, the "=" separator is used for simple (non-hierarchical) properties and the " " separator is used for complex (hierarchical) properties. As with the non-hierarchical properties format, any of the "=", ":", or " " separators can be used for both simple and composite properties.

```
http.server.worker.count = 8

file.serve {
    location      = /
    document.root = file:src/doc
    cache.ttl     = 3600
}
```

The hierarchical property format includes the same set of features as the non-hierarchical property format, including comments, escape codes and line continuation.

```
# A hierarchical property file that has comments,
# escaped characters, and line continuation.

\#a\ complexly\ named\ property\# = \{ a value with curly brackets \}

prop = a value with ## hash !! and excl

group.config {
    # Use of line continuation.
    files = file1.txt \
          , file2.txt \
          , file3.txt
}
```

Included files can be specified via the *"@"* directive. This directive takes an absolute URI, an absolute path, or a relative path.

```
# An HTTPS include directive.
@https://borasoftware.com/doc/examples/hprops.properties

# An absolute path include directive.
@/etc/balau/default-sites/default.site

# A relative path include directive.
@extra-sites/special.site
```

When an absolute or relative path is specified as in the second and third examples above, the URI type resolved should be the same as the URI of the property file that contains the include directive (this is performed by the implementation consuming the parsed file contents). For example, if the above example property file was supplied as a file URI, the absolute and relative path include directives would resolve to file URIs.

Include directives may also contain glob patterns.

```
# A globbed, relative path include directive.
@sites-enabled/*.site
```

Glob patterns are only supported by certain URI types (e.g. files and zip archives). It is the responsibility of the property file writer/consumer to ensure that globbed includes are only used for URI types that support them.

Parsing

#include <Balau/Lang/Property/PropertyParsingService.hpp>

Creating a hierarchical properties parser and parsing some input text involves a single line of code.

```
// The input URI that represents the source properties text (normally sourc
Resource::File input("somePropertyFile.properties");

// Call the parsing service.
Properties properties = PropertyParsingService::parse(input);
```

Printing the parsed properties AST back into text can be achieved via the *PropertyAstToString* visitor class. This is normally performed via the *PropertyNode toString* function.

```
// Pretty print the hierarchical properties AST back out to a string.
std::string propertiesText = toString(items);
```

Visiting

#include <Balau/Lang/Property/Util/PropertyVisitor.hpp>

Once the input properties text has been parsed into an AST, it can be visited by implementing the *PropertyVisitor* interface.

As an example, an extract from the *PropertyAstToString* class provided in the Balau library is given below.

```
class PropertyAstToString : public PropertyVisitor {
    public: void visit(Payload & payload, const Properties & object) overr
        for (auto & node : object.getNodes()) {
            node->visit(payload, *this);
        }
    }

    public: void visit(Payload & payload, const ValueProperty & object) ove
        auto & pl = static_cast<PropertyAstToStringPayload &>(payload);
        pl.writeIndent();
        pl.write(object.getName());
        pl.write(" = ");
        pl.write(object.getValue());
        pl.write("\n");
    }

    public: void visit(Payload & payload, const CompositeProperty & object
        auto & pl = static_cast<PropertyAstToStringPayload &>(payload);
        pl.writeIndent();
        pl.write(object.getName());
        pl.write(" {\n");
        pl.incrementIndent();

        for (const auto & node : object.getNodes()) {
            node->visit(payload, *this);
        }

        pl.decrementIndent();
        pl.writeIndent();
        pl.write("}\n");
    }

    // ... more visitor methods ...
};
```

When creating a custom properties AST visitor implementation, a quick way of achieving this is to copy the *PropertyAstToString* class and modify it to meet the requirements of the new visitor implementation.

Hierarchical format

The basic format of the hierarchical property format is the same as that of Java *.properties* files.

- Leading blanks of each line are stripped.
- Resulting lines that do not begin with a '#' or '!' character and that end with the '\' character are concatenated with the next line.
- Resulting lines that begin with a '#' or '!' character are comments.

- Property lines have a key and an optional value, separated by a key value separator.
- Key-value separators are ':', '=', or blanks.
- Blanks can surround ':' and '=' separators; these do not form part of the key and value.
- The entire remaining line after the separator and optional blanks is the value, including any trailing blanks.
- Special characters (blank, ':', '=', '#', '!', '\') can be used in keys by escaping them with a '\' prefix.
- Unrecognised escaped characters result in the '\' prefix being silently dropped.

The following additional rules add the hierarchical extension to the Java *.properties* file format.

- The '{', '}', and '@' characters must be escaped when used in property names and values.
- A non-escaped '{' character that follows a property separator denotes the start of a hierarchical property block.
- The '{' character of a hierarchical property block must immediately be followed by a line break.
- A matching '}' character must exist that denotes the block end. The '}' block end character must be placed on a line of its own.
- In between the '{' and '}' property block delimiters, zero or more child properties may be defined, including more hierarchical properties.
- A non-escaped '@' character at the start of a line denotes the start of an include directive. The URI (full, partial absolute path, or partial relative path) following the '@' character represents the include URI.
- Indentation of block closing '}' characters and include directive '@' characters is not relevant and is stripped during parsing.

Classes

All the classes are found in the *Balau::Lang::Property* namespace.

Class/enum	Description
PropertyToken	Language terminals enum
PropertyNode	Abstract base class of the language non-terminal AST nodes
PropertyScanner	The property scanner implementation
PropertyParser	The property parser implementation
PropertyParserService	Convenience class providing single function parsing.
PropertyVisitor	The tightly coupled AST visitor interface
PropertyAstToString	Property AST pretty printer

Data structures

The data structures used to hold the data generated from the Balau hierarchical properties parser are as follows.

Node classes can only exist within the context of an owning *Properties* instance that owns the parsed string.

```
///
/// Partial base class of nodes.
///
struct PropertyNode {
};

///
/// The outer structure. A single instance of this
/// struct represents the entire parsed properties text.
///
struct Properties : public PropertyNode {
    std::string text;
    std::vector<std::unique_ptr<PropertyNode>> nodes;
};

///
/// Partial implementation of a key-value node.
///
struct ValueProperty : public PropertyNode {
    std::string_view key;
    std::string_view value;
};

///
/// Partial implementation of a hierarchical node.
///
struct CompositeProperty : public PropertyNode {
    std::string_view key;
    std::vector<std::unique_ptr<PropertyNode>> nodes;
};

///
/// Partial implementation of an include node.
///
struct IncludePropertyNode : public PropertyNode {
    std::string_view text;
};

///
/// Partial implementation of a comment line node.
///
struct CommentPropertyNode : public PropertyNode {
    std::string_view text;
};
```

As the AST classes are views onto the original input text, the names and values of properties are string views onto the original text, including any line continuation / leading blank combinations. In addition, escaped characters are in their escaped form.

In order to obtain final name and value text, the *ValueProperty* AST class has *getName* and *getValue* methods, and the *CompositeProperty* AST class has a *getName* method. These methods will process name and value text into the final form.

Grammar

Notation

The following notation is used in the grammar.

Symbol	Meaning
=	definition of rule
()	grouping for precedence creation
*	zero or more repetitions
+	one or more repetitions
?	optional
|	choice separator
(^ .. | ..)	any except the content choice
"text"	literal string in terminal
// ..	comment

The choice separator has lowest notation precedence. All other notational entities have equal precedence.

Whitespace

A "\" character placed at the end of a line indicates line continuation. All non-escaped blanks (space/tab) occurring at the start of a line are semantically removed from property names /values that are broken up by line continuation. This is not represented in the grammar and thus occurs after parsing.

Explicit non-terminals

```
// Explicit non-terminals are the produced AST nodes.

S               = Properties

Properties      = Property*

Property        = Blank* (ValueProperty | ComplexProperty | Include | Comme
                  (LineBreak | LineBreak? EndOfFile)

ValueProperty   = Key (Assignment Value?)?

ComplexProperty = Key Assignment OpenCurly LineBreak Property* CloseCurly

Include         = Arobase
                  ( OpenCurly | CloseCurly | Arobase     | Colon | Equals
                  | Blank     | Hash       | Exclamation | Text  | BackSlas
```

```
Comment          = (Hash | Exclamation)
                   ( OpenCurly | CloseCurly | Arobase     | Colon | Equals
                   | Blank     | Hash       | Exclamation | Text  | BackSla
```

Implicit non-terminals

```
// Implicit non-terminals are assimilated into produced AST nodes.

Assignment       = ((Blank? (Equals | Colon) Blank?) | Blank)

Key              = KeyStart KeyCont

KeyStart         = Text            | EscapedOpenCurly | EscapedCloseCurly
                 | EscapedArobase  | EscapedColon     | EscapedEquals
                 | EscapedHash     | EscapedExcl      | EscapedBackSlash
                 | EscapedBlank    | EscapedChar      | (LineCont KeyCont)

KeyCont          = ( Text            | EscapedOpenCurly | EscapedCloseCurly
                   | EscapedArobase  | EscapedColon     | EscapedEquals
                   | EscapedHash     | EscapedExcl      | EscapedBackSlash
                   | EscapedBlank    | EscapedChar      | Hash
                   | Exclamation     | (LineCont KeyCont)
                   )*

Value            = ValueStart ValueCont

ValueStart       = Text            | EscapedOpenCurly | EscapedCloseCurly
                 | EscapedArobase  | EscapedColon     | EscapedEquals
                 | EscapedHash     | EscapedExcl      | EscapedBackSlash
                 | EscapedBlank    | EscapedChar      | Hash
                 | Exclamation     | CloseCurly       | Colon
                 | Equals          | Blank            | (LineCont ValueCont)

ValueCont        = ( Text            | EscapedOpenCurly | EscapedCloseCurly
                   | EscapedArobase  | EscapedColon     | EscapedEquals
                   | EscapedHash     | EscapedExcl      | EscapedBackSlash
                   | EscapedBlank    | EscapedChar      | Hash
                   | Exclamation     | CloseCurly       | Colon
                   | Equals          | Blank            | OpenCurly
                   | (LineCont ValueCont)
                   )*

LineCont         = EscapedLineBreak Blank*
```

Terminals

```
// The terminal strings in the definitions use \t, \r, \n, and \\
// placeholders in regular expressions (purple strings) to denote
// tab, carriage return, line feed, and BackSlash characters.

OpenCurly          = "{"
CloseCurly         = "}"
```

```
Arobase             = "@"
Colon               = ":"
Equals              = "="
Blank               = space | "\t"
LineBreak           = "\r\n|\n\r|\n|\r"
Hash                = "#"
Exclamation         = "!"
EndOfFile           = no further input available
Text                = "[^{}@:= \t\r\n#!\\]+"
BackSlash           = "\"

EscapedOpenCurly    = "\{"
EscapedCloseCurly   = "\}"
EscapedArobase      = "\@"
EscapedColon        = "\:"
EscapedEquals       = "\="
EscapedHash         = "\#"
EscapedExcl         = "\!"
EscapedBackSlash    = "\\"
EscapedBlank        = "\ " | "\\\t"
EscapedChar         = "\\[^{}:=#!\\ \t\r\n]"
EscapedLineBreak    = "\\(\r\n|\n\r|\r|\n)"
```

NETWORK

HTTP client

Overview

HTTP and HTTPS clients that use the Boost Asio and Beast libraries.

The clients currently provide synchronous GET, HEAD, and POST calls.

Quick start

#include <Balau/Network/Http/Client/HttpClient.hpp>
#include <Balau/Network/Http/Client/HttpsClient.hpp>

Construction

In order to create a client, specify the host to the client constructor.

```
// Create an HTTPS client with default SSL port 443.
HttpsClient client("borasoftware.com");
```

The default user agent used in the client is "*Balau <version>*", where *<version>* is the version of the Balau library. The HTTP version in the above client is version 1.1.

If the required port, user agent and/or HTTP version are different to the default values, they can be specified in the constructor

```
// Create an HTTP client with port 12345.
HttpClient client1("example.com", 12345);

// Create an HTTP client with port 12345 and user agent "Anon".
HttpClient client2("example.com", 12345, "Anon");

// Create an HTTP client with port 12345, user agent "Anon", and HTTP vers
HttpClient client3("example.com", 12345, "Anon", "1.0");
```

If the scheme in the URL should determine whether an HTTP or HTTPS client should be created, the *HttpClient::newClient* static method can be used. In this API, the optional port number should also be included in the URL.

```
// Create HTTP or HTTPS clients, according to the supplied URL.
auto client4 = HttpClient::makeClient("https://borasoftware.com");
auto client5 = HttpClient::makeClient("http://example.com:12345");
```

The *HttpClient::newClient* static method also accepts an optional user agent and HTTP version.

```
// Create a client with a custom user agent.
auto client6 = HttpClient::makeClient("https://borasoftware.com", "Anon");

// Create a client with a custom user agent and HTTP version 1.0.
auto client7 = HttpClient::makeClient("http://example.com:12345", "Anon", '
```

Usage

In order to perform synchronous GET, HEAD, and POST calls, the *get*, *head*, and *post* methods can be used.

```
// Perform a GET request.
CharVectorResponse response1 = client.get("/");

// Perform a HEAD request.
EmptyResponse response2 = client.head("/");

// Perform a POST request.
const std::string body = generateBody();
CharVectorResponse response3 = client.post("/api/execute", body);
```

The GET and POST requests return a *CharVectorResponse*, which contains response headers and a character vector body. The HEAD request returns an *EmptyResponse*, which only contains response headers.

HTTP server

Overview

An HTTP and WebSocket server that uses the Boost Asio and Beast libraries.

A Balau HTTP server instance contains two trees of web applications, one for HTTP and another for WebSockets. These trees allow the server to handle HTTP requests differently, according to the HTTP request paths. WebSocket clients are also connected to different WebSocket applications, based on the HTTP request path active during the WebSocket upgrade.

Complex routing and handling of HTTP requests by a single HTTP server may be created via the creation of a sophisticated HTTP web application tree. Similarly, the specification of the WebSockets application tree allows multiple WebSockets applications to be provided by the same HTTP server.

There are two ways to use the HTTP server:

- specify server configuration parameters and web application trees directly via one of the HTTP server's constructors;
- specify HTTP server and web application tree configuration via environment configuration.

The first type of usage is a more traditional approach to application development, where the web application framework is fixed at code creation.

The second type of usage allows different web application configurations to be specified for different environments whilst using the same pre-compiled application. The resulting environment configurations provide the configurable parameters of the main HTTP server and the web applications, modifiable independently of application compilation.

Quick start

#include <Balau/Network/Http/Server/HttpServer.hpp>

Environment configuration: http.server

Hardwired

There are two HTTP server constructors available for direct usage.

The first constructor accepts global server configuration and both predefined HTTP and WebSocket handlers. By specifying routing web application handlers, this constructor also allows full HTTP and WebSockets application trees to be specified. If no WebSockets handler or application tree is required, a null handler may be specified.

The second constructor accepts global server configuration and a document root. This constructor is useful in order to create a simple file serving HTTP server.

Both these constructors may be used within an injector provider if required, in order to integrate them into the application's bindings tree.

Please refer to the HTTP server source code or the Balau API documentation to see the exact signatures of the two constructors.

Injected

The injectable constructor of the HTTP server takes an *EnvironmentProperties* instance. When supplied from the injector, this is bound to the root *EnvironmentProperties* binding with name "*http.server*".

```
HttpServer(std::shared_ptr<System::Clock> clock,
           std::shared_ptr<EnvironmentProperties> configuration,
           bool registerSignalHandler = true);
```

The clock instance should be provided by another binding in the injector's application configuration. The *http.server.register.signal.handler* boolean binding is defaulted to *true* via the Balau environment configuration specifications. If you intend to set up your own signal handlers which will manage the HTTP server's shutdown, your environment configuration should have a *http.server.register.signal.handler* root property set to *false*.

The above constructor will be called by the injector when a suitable binding is added to the application configuration.

```
// Example application configuration for the HTTP server.
class AppConfig : public Balau::ApplicationConfiguration {
    public: void configure() const override {
        bind<Balau::System::Clock>().toSingleton<Balau::System::SystemClocl
        bind<Balau::Network::Http::HttpServer>().toSingleton();
    }
};
```

Once the HTTP server binding has been specified in the application configuration, an environment configuration entry for the HTTP server will need to be created, with name *"http.server"* binding. The contents of the HTTP server's environment configuration will depend on the usage requirements and the required web application trees. Please refer to the http.server environment configuration documentation for more information.

During instantiation, the HTTP server will create an HTTP routing web application and a WebSockets routing web application, and will populate them with instances of the web applications specified in the HTTP server's environment configuration.

Below is an example of environment configuration for an HTTP server configured with file server and email sender HTTP web applications.

```
http.server {
    info.log     = file:logs/access.log
    error.log    = file:logs/error.log
    server.id    = My server
    worker.count = 8
    listen       = 127.0.0.1:8080

    # Mime types specified in a different file.
    @file:mime.types.hconf

    http {
        files {
            location = /
            root     = file:www
        }

        email.sender {
            location   = /ajax/send
            host       = smtp.example.com
            port       = 465
            user       = email-sender
            subject    = MESSAGE RECEIVED
            from       = enquiry@example.com
            to         = enquiry@example.com
            user-agent = Balau

            # These will be served by the file serving web app.
            success    = /success.html
            failure    = /failure.html

            # These parameters should match the POST request form data.
            parameters {
                Name  = 1
                Email = 2
                Tel   = 3
                Body  = 4
            }
        }
    }
}
```

In order to create a binding in the injector for the environment configuration, an *EnvironmentConfiguration* instance for the environment resource(s) will need to be created. In addition to a URI for the environment configuration, the instance should be supplied with a

URI specifying the Balau environment specifications. These specifications will be used to provide type information and any default values not provided in the configuration.

```
///////////// Create the application injector. /////////////

// A file pointing to the environment configuration given above.
auto env = Resource::File("path/to/env/env.hconf");

// A file pointing to the environment's credentials configuration.
// This is required for the email.sender user password.
auto creds = Resource::File("path/to/env/creds.hconf");

// A file pointing to the Balau environment specifications.
auto specs = Resource::File("path/to/specs/balau.thconf");

// Create the injector configurations.
auto appConf = AppConfig();
auto envConf = EnvironmentConfiguration({ env, creds }, { specs });

// Create the injector.
auto injector = Injector::create(appConf, envConf);
```

Configuration

This section discusses the HTTP server configuration in more detail.

Main configuration

The HTTP server main configuration properties are supplied within a composite *http.server* composite property. This property forms a container of all configuration required by the HTTP server and web applications, with the exception of credentials properties.

The hierarchical structure of the main configuration also specifies the HTTP and WebSocket web application hierarchies. During instantiation, the HTTP server will instantiate the appropriate web applications, according to the environment configuration.

The structure of the main configuration takes the form of a set of HTTP value properties, plus a hierarchical set of composite *location* properties. Each *location* property specifies a location, the web application handler, and the web application's configuration.

The following is an example of a simple HTTP server main configuration that has a file serving web application and an email sending web application.

```
http.server {
    ######### General server properties #########

    logging.ns    = http.server
    info.log      = file:logs/access.log
    error.log     = file:logs/error.log
    server.id     = My server
    worker.count = 8
    listen        = 127.0.0.1:8080

    # Include mime types file.
    @file:mime.types.hconf

    ############ HTTP request filters ###########

    filters {
        # TODO
    }

    ###### HTTP web application properties ######

    http {
        files {
            location = /
            root     = file:www
        }

        email.sender {
            location   = /ajax/send
            host       = smtp.example.com
            port       = 465
            user       = email-sender
            subject    = MESSAGE RECEIVED
            from       = enquiry@example.com
            to         = enquiry@example.com
            user-agent = Balau

            # These will be served by the file serving web app.
            success    = /success.html
            failure    = /failure.html

            # These parameters should match the POST request form data.
            parameters {
                Name  = 1
                Email = 2
                Tel   = 3
                Body  = 4
            }
        }
    }

    ### WebSockets web application properties ###

    ws {
        # TODO
    }
}
```

Credentials management

HTTP server web application credentials are supplied in the same hierarchy as the main web application configuration. In order to physically separate confidential credentials from the main environment configuration, a parallel tree may be created that contains only credentials information. The two configuration trees will then be merged together by the injector's environment configuration logic, resulting in a single tree.

In order to merge the two configuration files, a single *EnvironmentConfiguration* instance should be created.

```
auto env = EnvironmentConfiguration({ env, creds }, { specs });
```

HTTP web applications

Overview

This documentation chapter contains information on the HTTP web application framework and the predefined HTTP web applications currently available in the Balau library. Developers may also define their own HTTP web applications.

Framework

Creation

The HTTP web application framework is based around the *HttpWebApp* base class. All HTTP web applications are derived from this base class. HTTP web applications must be registered with the HTTP server framework. Predefined Balau HTTP web applications are automatically registered at application startup. Custom HTTP web applications may be registered by calling *HttpWebApp::registerHttpWebApp<WebAppT>(name)* before creating the application injector.

Each HTTP web application implements the three request call methods:

```
void handleGetRequest(HttpSession & session,
                      const StringRequest & request,
                      std::map<std::string, std::string> & variables) over

void handleHeadRequest(HttpSession & session,
                       const StringRequest & request,
                       std::map<std::string, std::string> & variables) over

void handlePostRequest(HttpSession & session,
                       const StringRequest & request,
                       std::map<std::string, std::string> & variables) over
```

HTTP session

The *HttpSession* passed during a request call references the active HTTP session for the request in progress. The session object is active on a single thread and may handle multiple keep-alive round trips. The session object provides access to:

- the HTTP server's main configuration;
- the remote IP address for logging;
- the client session;
- the cookies sent in the request;

- the sendResponse method used to send the response.

Client session

The client session referenced within the HTTP session is a long lived session that is obtained on each request by examining the session cookie sent by the client. The name of this session cookie may be specified in the HTTP server's global configuration. By default, the name of the session cookie is *session*.

The client session Id is present for web applications to use in order to allow stateful sessions such as logins and carts.

Request object

The request object passed during a request call is the full Boost Beast request. Web applications can access request fields and other information available from the Beast request message API.

Request variables

The request variables map passed during a request call are variables that are created and consumed by filters and web applications during the request. They are not related to the request HTTP fields. An example of request variables can be seen in the *redirections* HTTP web application, which creates temporary request variables named *$1*, *$2*, *$2*, etc. for regular expression groupings in the redirection matches.

Configuration

Each web application must have a *location* parameter in its configuration. The value of this parameter is a space delimited set of location prefixes that the web application will handle. During instantiation, the HTTP server will read this parameter on each web application's configuration and use the location prefixes within to construct the request routing.

Web applications

File server

#include <Balau/Network/Http/Server/HttpWebApps/FileServingHttpWebApp.hpp>

Environment configuration: files

The file serving HTTP web application serves files from a directory on the local file system.

See the files environment configuration for details on how to configure the file serving HTTP web application.

Email sender

#include <Balau/Network/Http/Server/HttpWebApps/EmailSendingHttpWebApp.hpp>

Environment configuration: email.sender

The email sending HTTP web application sends an email with a body generated from the form parameters of a POST request. This can be useful for creating a contact page.

See the email.sender environment configuration for details on how to configure the email sending HTTP web application.

Redirector

#include <Balau/Network/Http/Server/HttpWebApps/RedirectingHttpWebApp.hpp>

Environment configuration: redirections

The redirecting HTTP web application performs 301 or 302 redirections for specified locations.

See the redirections environment configuration for details on how to configure the redirecting HTTP web application.

Canned

#include <Balau/Network/Http/Server/HttpWebApps/CannedHttpWebApp.hpp>

Environment configuration: canned

The canned HTTP web application serves fixed responses for GET, HEAD, and POST requests.

See the canned environment configuration for details on how to configure the canned HTTP web application.

Failing

#include <Balau/Network/Http/Server/HttpWebApps/FailingHttpWebApp.hpp>

Environment configuration: failing

The failing HTTP web application returns HTTP 404 responses for all requests.

See the failing environment configuration for details on how to configure the failing HTTP web application.

Routing

#include <Balau/Network/Http/Server/HttpWebApps/RoutingHttpWebApp.hpp>

Environment configuration: n/a

The current routing web application may be replaced by a regular expression based routing web application in the future.

The routing HTTP web application contains a trie data structure used to route HTTP requests to different HTTP web applications.

The routing HTTP web application is normally used implicitly via environment configuration. The HTTP server will instantiate a routing HTTP web application when creating the HTTP web application tree. When defining the HTTP server HTTP web applications via environment configuration, it is thus not necessary to explicitly define a routing HTTP web application.

The routing HTTP web application may be explicitly used when manually building an HTTP web application routing tree in code. Simple examples of this are available in the Balau unit tests. For example, one of the email sender unit tests defines the following routing HTTP web application.

```
// Create the routing HTTP web app, specifying the root node.
RoutingHttpWebApp::Routing routing(routingNode<FailingHttpWebApp>(""));

// Add the email sender handler at path "/1/send-message".
routing.add(
      routingNode("1")
    , RoutingHttpWebApp::Node::child(
        RoutingHttpWebApp::Value(
            "send-message", emailHandler, emailHandler, emailHandler
        )
    )
);
```

In order to facilitate the manual building of routing trees, the *RoutingHttpWebApp.hpp* header contains the following helper methods and type.

Name	Description
routingNode	Make a routing node (3 overloads).
RoutingHttpWebApp::Node::child	Add a child of the child being added, plus descendants of the child (2 overloads).
RoutingHttpWebApp::Value	The tuple type representing a routing node.

WebSocket app framework

Overview

This documentation chapter is pending.

Quick start

#include <Balau/Network/Http/Server/WsWebApp.hpp>

This documentation chapter is pending.

SYSTEM

Clock

Overview

The Balau clock infrastructure consists of a base *Clock* interface and a single *SystemClock* implementation.

The *Clock* interface is both a convenient API for clock functions, but more importantly is a way to allow the injection of a test clock implementation into production code. This allows the manipulation of the clock from within test methods in order to simulate changes in time.

The clock API uses both std::chrono and the embedded Hinnant date library in the *ThirdParty* folder.

Quick start

#include <Balau/System/Clock.hpp>
#include <Balau/System/SystemClock.hpp>

Clock binding

When developing an application, a *Clock* binding should be added to the application's main injector configuration. Then classes that require the clock API can have the clock injected into them via their injectable constructor.

```
// The application's main injector configuration.
class Configuration : public ApplicationConfiguration {
    public: void configure() const override {
        // The production clock implementation.
        bind<Clock>().toSingleton<SystemClock>();

        // ... more binding declarations ...
    }
};

// An injector aware class that requires the clock API.
class AService {
    std::shared_ptr<Clock> clock;
    // ... more dependencies ...

    BalauInjectConstruct(
          AService
        , clock
        // ... more injectables ...
    );
};
```

In order to test the class, a test clock may be created by deriving from either the *Clock* interface directly or extending the *SystemClock* class and reimplementing one or more of the methods. An instance of this test clock can then be injected (manually or via a test injector) into the class to be tested.

Clock API

The clock API is currently quite brief. More features will be added in future releases.

The following methods are declared.

Method name	Description
now	Get the current time point.
today	Get the current date.
nanotime	Get the current time in nanoseconds since the unix epoch.
millitime	Get the current time in milliseconds since the unix epoch.

Sleep utilities

Overview

The *Sleep* namespace class contains static methods that cause the current thread to sleep for the specified duration.

Quick start

#include <Balau/System/Sleep.hpp>

The following static methods are declared.

Method name	Description
sleep	Sleep for the indicated number of seconds.
milliSleep	Sleep for the indicated number of milli-seconds.
microSleep	Sleep for the indicated number of micro-seconds.
nanoSleep	Sleep for the indicated number of nano-seconds.

Thread naming

Overview

The *ThreadName* namespace class provides a way to set a thread local string that is used by the logging system for the name of the current thread.

The implementation is very simple and only allows access to the thread name from the thread itself. *ThreadName* exists mainly to support logging, but may be used for other applications that do not need to access the names of other threads.

It is the responsibility of the end developer to ensure the same name is not used for multiple threads.

Quick start

#include <Balau/System/ThreadName.hpp>

The *ThreadName* namespace class provides two static methods, one for setting the current thread's name and another to get the current thread's name.

```
// Set the current thread's name.
ThreadName::setName("Main");

// Get the current thread's name.
const auto & name = ThreadName::getName();
```

Once a thread's name has been set, the logging system's *%thread* format specifier will print the thread name instead of the native platform's thread id.

UTIL

Compression utilities

Overview

This header provides user friendly compression utility functions defined within the *Balau::Util::Compression* namespace, and the *Unzipper* / *Zipper* class pair for zip file access and mutation.

Quick start

Gzip utilities

The *Gzip* class is a namespace class that contains functions to deflate and inflate data between files, strings, and streams.

The following static methods are currently defined.

Function name	Description
gzip	Gzip the input file/string/ostream to the specified output file.
gunzip	Gunzip the input file to the specified output file/string/istream.

Each method is overloaded to allow file, string, and stream input/output.

Zipper and Unzipper

These classes provide reading and writing functionality for zip files. *Unzipper* provides a reading API for immutable zip files. *Zipper* extends the reading API *Unzipper* with mutation functionality.

The implementation uses LibZip as the backend implementation. Consequently, the read and write APIs reflect the functionality of the backend library. Refer to the Balau Unzipper and Zipper API documentation for more information on these classes.

In addition to the *Zipper* and *Unzipper* classes, the *ZipFile* and associated *ZipEntry* resource classes use the *Unzipper* class to provide a recursive iterator into zip archives. For more information, refer to the ZipFile and ZipEntry documentation.

Date-time utilities

Overview

This header provides user friendly date/time utility functions, defined within the *Balau::Util::DateTime* namespace.

Balau includes a copy of the Hinnant date library in the *ThirdParty* folder. The library is used by Balau and can be used directly in applications that use Balau. Documentary references to this library refer to the library as the *HH date library*.

Quick start

#include <Balau/Util/DateTime.hpp>

The following functions are currently implemented. More information is available on the API documentation page.

Function name	Description
toString	Format the time point as a string with the specified format.
toDuration	Create a duration from the supplied string which is in the specified format.

File utilities

Overview

This header provides user friendly file utility functions, defined within the *Balau::Util::Files* namespace.

Quick start

#include <Balau/Util/Files.hpp>

The following functions are currently implemented. More information is available on the API documentation page.

Function name	Description
copy	Copy the contents of the source file into the destination file.
toLines	Read all lines of text of the specified file into a string vector.
readToString	Read the specified file into a string.
readToVector	Read the specified file into a character vector.

Memory utilities

Overview

This header provides memory utility functions, defined within the *Balau::Util::Memory* namespace.

Quick start

#include <Balau/Util/Memory.hpp>

The following utilities are currently implemented. More information is available on the API documentation page.

Pointer containers

These functions provide variadic conversions from a set of input arguments to a vector of pointer containers, via a supplied transform function. Functions are available for shared and unique pointer containers.

There are two versions of the functions. The first version works with a transform function that transforms an argument into a single pointer. The second version works with a transform function that transforms an argument into a vector or pointers.

These functions are quite specialised and specific to the Balau library's internal implementation. For example, the *makeSharedV* function is used in the injector implementation in order to create the binding builders from each supplied configuration object in the variadic *createBindings* call.

```
auto builders = Memory::makeSharedV<ApplicationConfiguration, Impl::Bindin
    [] (const ApplicationConfiguration & conf) { return conf.createBuilder
    , conf ...
);
```

These functions may however be useful when a similar variadic transform functionality is required in application code.

Function name	Description
makeShared	Create a vector of shared pointers after transforming the arguments (scalar transform version).
makeSharedV	Create a vector of shared pointers after transforming the arguments (vector transform version).
makeUnique	Create a vector of unique pointers after transforming the arguments (scalar transform version).
makeUniqueV	Create a vector of unique pointers after transforming the arguments (vector transform version).

Pretty printing

Overview

This header provides utilities for printing numeric values in different formats, defined within the *Balau::Util::PrettyPrint* namespace.

Quick start

#include <Balau/Util/PrettyPrint.hpp>

The following pretty print functions are currently implemented. More information is available on the API documentation page.

Function name	Description
fixed	Print the value in fixed notation.
scientific	Print the value in scientific notation.
metricPrefix	Print the value with a metric prefix.
binaryPrefix	Print the value with a binary prefix. This is similar to a metric prefix, but uses 2^10 (1024) as the divisor.
byteValue	Returns a string containing the supplied byte value in terms of B/KB/MB /GB etc.
duration	Pretty print the duration.
printHexBytes	Print the value as bytes in hexadecimal.

Random number generators

Overview

Convenience wrappers around the C++11 random number generator library.

The implementation is based around a single *RandomNumberGenerator* template class. A set of aliases are provided for each of the random number generator configurations.

The defined random number configurations are given in the table towards the end of this chapter.

Quick start

#include <Balau/Util/Random.hpp>

Construction

To create a *RandomNumberGenerator*, choose the configuration you require and supply the boundaries within which the generator will generate numbers.

```
// Create a uniform double random number generator that will generate
// double precision floating point numbers between 0 and 10.
UniformDouble random(0, 10);
```

There is a second constructor available that takes an integer seed value in order to allow for repeatability (useful for testing).

```
// Create a uniform double random number generator with a seed.
UniformDouble random(0, 10, 12345);
```

Usage

Once constructed, random numbers can be obtained from the generator by calling its *operator ()*.

```
// Generate some random numbers.
double a = random();
double b = random();
double c = random();
```

Generator types

This section lists the different types of generated that are defined.

Uniform distribution

Name	Description
UniformDouble	A uniform distribution, double precision floating point random number generator.
UniformFloat	A uniform distribution, single precision floating point random number generator.
UniformInt32	A uniform distribution, 32 bit signed integer floating point random number generator.
UniformInt64	A uniform distribution, 64 bit signed integer floating point random number generator.
UniformUInt32	A uniform distribution, 32 bit signed integer floating point random number generator.
UniformUInt64	A uniform distribution, 64 bit signed integer floating point random number generator.

Normal distribution

Name	Description
NormalDouble	A normal distribution, double precision floating point random number generator.
NormalFloat	A normal distribution, single precision floating point random number generator.

Templated types

These templated aliases require the type *T* to be specified. Most developers will not require these.

Name	Description
UniformReal	A uniform distribution, floating point random number generator.
UniformInt	A uniform distribution, integer random number generator.
Normal	A normal distribution random number generator.
LogNormal	A log normal distribution random number generator.
Gamma	A gamma distribution random number generator.
ChiSquared	A chi squared distribution random number generator.
Cauchy	A Cauchy distribution random number generator.
FisherF	A Fisher F distribution random number generator.
StudentT	A Student T distribution random number generator.
Binomial	A discrete binomial distribution random number generator.
Geometric	A geometric distribution random number generator.
NegativeBinomial	A negative binomial distribution random number generator.
Poisson	A Poisson distribution random number generator.
Exponential	An exponential distribution random number generator.
Weibull	A Weibull distribution random number generator.
ExtremeValue	An extreme value distribution random number generator.
Discrete	A discrete distribution random number generator.
PiecewiseConstant	A piecewise constant distribution random number generator.
PiecewiseLinear	A piecewise linear distribution random number generator.

Apart from the numeric template type, the use of the templated generators is the same as the fully typed generators.

```
// Create a double precision Poisson random number generator.
Poisson<double> random(0, 10);

double a = random();
double b = random();
double c = random();
```

Stream utilities

Overview

This header provides user friendly stream utility functions, defined within the *Balau::Util::Streams* namespace.

Quick start

#include <Balau/Util/Streams.hpp>

The following functions are currently implemented. More information is available on the API documentation page.

Function name	Description
consume	Consume all the data from the supplied input stream into the supplied output stream.
readLinesToVector	Read all lines of text from the supplied input stream into a vector.

String Utilities

Overview

This header provides user friendly UTF-8 and UTF-32 string utility functions, defined within the *Balau::Util::Strings* namespace.

UTF-16 versions of the utilities are not current implemented. Pull requests with such new functionality are welcome.

These utility functions provide more coarse grained functionality compared to that provided by ICU, Boost, and the C++ standard library string manipulation calls.

The utilities use and return *std::string_view* objects where possible. In order to support multiple string types, a two stage typename deduction / type conversion approach is used for many of the functions.

Quick start

#include <Balau/Util/Strings.hpp>

The following utilities are currently implemented. More information is available on the API documentation page.

https://borasoftware.com/doc/balau/latest/api/structBalau_1_1Util_1_1Strings.html

Each function covers UTF-8 and UTF-32 encodings and is available in several forms.

Examination

Function name	Description
startsWith	Does the first string start with the second string/character/code point?
endsWith	Does the first string end with the second string/character/code point?
contains	Does the first string contain the second string/character/code point?
occurrences	How many non-overlapping occurrences of the second string/regular expression are found in the first string?
equalsIgnoreCase	Ignoring case, is the first string equal to the second string?
lineLengths	Given the supplied multi-line text string and an optional line break regular expression, determine the lengths of the lines in bytes.
lastIndexOf	Get the character/code point position in the first string of the last index of the second string.
codePointCount	Count the number of code points in the supplied string.

Mutation

Function name	Description
toUpper	Convert the string to uppercase.
toLower	Convert the string to lowercase.
append	Appends count times the supplied character/code point to the supplied string.
padLeft	Left pad the string up to the specified width in code points, using the supplied character/code point.
padRight	Right pad the string up to the specified width in code points, using the supplied character/code point.
trim	Trim whitespace from the beginning and end of the string.
trimLeft	Trim whitespace from the beginning of the string.
trimRight	Trim whitespace from the end of the string.
replaceAll	Replace all occurrences of the specified string/regular expression with the specified replacement.

Manipulation

Function name	Description
join	Join the strings together, separated by the supplied delimiter.
prefixSuffixJoin	Join the strings together, prefixing each string with the prefix and suffixing each string with the suffix.
split	Split the string on each of the occurrences of the specified string/regular expression delimiter.

Vector utilities

Overview

This header provides user friendly utility functions, defined within the *Balau::Util::Vectors* namespace.

Quick start

#include <Balau/Util/Vectors.hpp>

The following utilities are currently implemented. More information is available on the API documentation page.

Appending

Function name	Description
append	Appends the source vector to the destination vector.
pushBack	Populate an existing vector via emplace back of multiple elements.

The *pushBack* function allows variadic emplace back to be performed in a single function.

Conversion

These functions convert vectors of characters to strings and strings to vectors of characters.

Function name	Description
charsToString	Convert the characters in the supplied char/char16_t/char32_t vector to a UTF-8/UTF-16/UTF-32 string.
toCharVector	Convert the supplied UTF-8/UTF-32 string to a char/char32_t vector.
toStringVector	Convert the supplied vector to a vector of UTF-8 strings, by calling *toString* on each object.
toString32Vector	Convert the supplied vector to a vector of UTF-32 strings, by calling *toString32* on each object.

Miscellaneous utilities

Introduction

This chapter documents small, miscellaneous types and utilities that are provided in Balau.

Assert

#include <Balau/Dev/Assert.hpp>

The *Assert* namespace class contains a set of runtime assertions used for development purposes. The assertions use the standard *assert()* call.

Enums

#include <Balau/Util/Enums.hpp>

The *Enums* namespace class currently contains a single static method:

```
///
/// Convert the strongly typed enum to its underlying integer.
///
template <typename E>
static auto toUnderlying(E e) noexcept -> typename std::underlying_type<E>
    return static_cast<typename std::underlying_type<E>::type>(e);
}
```

The *toUnderlying* method provides a clear indication in source code that the underlying integer value of the enum class is being obtained.

Hashing

#include <Balau/Util/Hashing.hpp>

The *Hashing* class is a namespace class used to hold hashing functions. There are three versions of each function, one accepting a *File*, another accepting a *string*, and a third accepting an *istream*.

The following hash algorithms are supported:

- SHA-256;
- SHA-3;
- Keccak;
- SHA-1;

- MD5;
- CRC32.

Macros

#include <Balau/Util/Macros.hpp>

The *Macros.hpp* header file contains some low level macros used in Balau.

OnScopeExit

#include <Balau/Type/OnScropeExit.hpp>
#include <Balau/Type/MoveableOnScropeExit.hpp>

The two classes *OnScopeExit* and *MovableOnScopeExit* provide RAII style management containers. The *MovableOnScopeExit* version is movable out of a scope without triggering the stored function.

An instance of *OnScopeExit* is created by supplying a function or lambda expression to the constructor. This function will be run from the destructor of the stack based object when the enclosing scope exits.

```
// Example usage of OnScopeExit class.

{
    Balau::OnScopeExit cleanUp([this] () { runCleanup(); })

    // ... more code ...

} // runCleanup() will be run here.
```

One interesting property of the *MovableOnScopeExit* implementation is that instances may be moved out of the scope via the move constructor. This allows a *MovableOnScopeExit* instance to be created inside a function and then returned to the caller on the stack, ready for destruction when the caller's scope exits.

The *OnScopeExit* and *MovableOnScopeExit* implementations are small classes, each class consisting of approximately 20 lines of code. Internally, the *OnScopeExit* implementation is based on a *std::function* stack based internal object and the *MovableOnScopeExit* implementation is based on a heap based internal object, managed inside a *std::unique_ptr*.

UUID

#include <Balau/Type/UUID.hpp>

The *UUID* class provides a convenient API for generating and manipulating UUIDs.

User

#include <Balau/Util/User.hpp>

The *User* class is a namespace class destined for functions that provide information on operating system users.

Currently, the *User* class contains a single *getHomeDirectory* function that makes a best effort attempt at returning the user's home directory for each supported platform.

App

#include <Balau/Util/App.hpp>

The *App* class is a namespace class destined for functions that provide general information for a running application.

The following functions are currently defined.

```
struct App {
    static File getUserApplicationDataDirectory(const std::string & appGro
    static File getGlobalApplicationDataDirectory(const std::string & appG
    static File getUserApplicationConfigDirectory(const std::string & appG
    static File getGlobalApplicationConfigDirectory(const std::string & ap
    static File getApplicationRuntimeDataDirectory(const std::string & app
};
```

The *appGroup* and *appName* strings provide the application's group name and identification name. These are used in the construction of the directory paths. The paths returned from these functions depend on the platform. Depending on the platform, the returned paths depend on the program binary's location, the user's identification, ${XDG_DATA_HOME}, ${HOME}, ${XDG_CONFIG_HOME}, and/or ${XDG_RUNTIME_DIR} environment variables, %USERPROFILE%, %HOMEDRIVE%, %HOMEPATH%, and/or GetTempPath() on Windows. See the API documentation for more details.

Note that these functions are not yet implemented for platforms other than Unix like platforms. Pull requests for other platforms are welcome.

COMMUNITY

Building Balau

This chapter discusses the preparatory steps, configuring, and building Balau.

Defaults

CMAKE_PREFIX_PATH

If the *CMAKE_PREFIX_PATH* is not set, the Balau CMakeLists.txt file defaults it to the *${HOME}/usr* folder. This setup allows the ICU, Boost, and optionally libzip dependencies to be installed in *${HOME}/usr*, to be picked up during CMake configuration automatically.

If the dependencies are located elsewhere, the *CMAKE_PREFIX_PATH* should be set, as discussed in the *CMake variables* section below.

CMAKE_INSTALL_PREFIX

If the *CMAKE_INSTALL_PREFIX* is not set, the Balau CMakeLists.txt file defaults it to the *${HOME}/usr* folder. If the library should be installed elsewhere (such as to */usr*), the *CMAKE_INSTALL_PREFIX* should be set, as discussed in the *CMake variables* section below.

Options

Balau contains a number of optional components that can be enabled/disabled during CMake configuration.

The currently available options are detailed in the table below.

Option	Default	Description
ALAU_ENABLE_ZLIB	ON	Enable ZLib library wrappers for gzip compression support.
ALAU_ENABLE_ZIP	ON	Enable LibZip library wrappers (Zipper and Unzipper).
LAU_ENABLE_CURL	ON	Enable use of Curl library (email sending web app).
LAU_ENABLE_HTTP	ON	Enable use of HTTP components (disabled for Boost < 1.68.0).

By default, ZLib and LibZip compression components are enabled, and the Curl components are not. Enabled components will require the corresponding library development files to be present during the build, as described in the dependencies section below.

Dependencies

In addition to the C++ standard library, Balau relies on two third party libraries and three utility libraries.

The first dependency is ICU, which provides Unicode support. ICU version **60.2** is the currently specified version in the CMakeLists.txt file.

The second dependency is the Boost library. Boost version **1.68.0** is the currently specified version in the CMakeLists.txt file.

The three utility library dependencies are *zlib*, *libzip* and *curl*. These libraries should be installed via your distribution's standard packaging system.

The only other dependencies used are standard dependencies on each supported platform.

Utility libraries

Debian/Ubuntu

The following command is for Ubuntu 18.04 and OpenSSL.

```
sudo apt install zlib1g-dev libzip-dev libcurl4-openssl-dev libssl-dev
```

RPM based distributions

Fedora 28 / 29

```
sudo yum install zlib-devel openssl-devel libcurl-devel libzip-devel
```

RHEL CentOS

Balau can be built on RHEL5 / CentOS v5 of later via the developer toolset 6 or later.

```
sudo yum install zlib-devel openssl-devel libcurl-devel
```

The supplied version of libzip on these distributions is *0.10.1*. A recent version of libzip must thus be compiled and installed. The following commands will download, build, and install libzip version *1.5.1* into the *${HOME}/usr* directory (this directory will be referenced later via *CMAKE_PREFIX_PATH*).

```
mkdir Libs
cd Libs
wget https://libzip.org/download/libzip-1.5.1.tar.gz
tar zxvf libzip-1.5.1.tar.gz
cd libzip-1.5.1
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=${HOME}/usr ..
make
make install
```

To install the developer toolset on CentOS, the *SCL* command can be used. DTS7 may be installed via the following command:

```
scl enable devtoolset-7 bash
```

RHEL / CentOS also ship with old versions of CMake (*2.8*). As Balau requires at least CMake version *3.10.2*, a suitable build of CMake must be available. Please refer to the CMake website for more information.

ICU

Linux

Download the ICU version 60.2 sources here.

To configure the ICU sources on Linux, unzip the sources and run the following commands in the unzipped ICU source directory. If you intend to use a different install prefix that the Balau default, the *--prefix* option should be set to the required path.

```
cd /path/to/icu/code
cd source

# Replace ${HOME}/usr to alternative path if required.
./runConfigureICU Linux --enable-static    \
                        --disable-shared   \
                        --disable-renaming \
                        --prefix=${HOME}/usr
```

Then add the necessary ICU defines in the *source/common/unicode/uconfig.h* file. These additionally configure ICU's build.

```
#define U_USING_ICU_NAMESPACE          1
#define UNISTR_FROM_CHAR_EXPLICIT      explicit
#define UNISTR_FROM_STRING_EXPLICIT    explicit
#define U_NO_DEFAULT_INCLUDE_UTF_HEADERS 1
#define U_HIDE_OBSOLETE_UTF_OLD_H      1
#define ICU_NO_USER_DATA_OVERRIDE      1
#define U_DISABLE_RENAMING             1
// Linux/OSx platforms also use:
#define U_CHARSET_IS_UTF8              1
```

Then build and install ICU.

```
make CXXFLAGS='-std=c++17 -g -o2 -fPIC' -j4
make install
```

Windows

This section will be filled in when the Windows port has been completed.

Boost

Linux

Download the Boost version 1.68.0 sources here.

To configure the Boost sources on Linux, unzip the sources and run the following commands in the unzipped Boost source directory. If you intend to use different install prefixes than the Balau default, the *--prefix* and *--with-icu* options should be set to the required path.

```
# Replace ${HOME}/usr occurrences to alternative paths if required.
./bootstrap.sh --with-icu=${HOME}/usr --prefix=${HOME}/usr
```

Then build and install Boost.

```
./b2 -j4
./b2 install
```

Windows

This section will be filled in when the Windows port has been completed.

CMake variables

Balau relies on two CMake variables in order to find its dependencies and to specify where to install itself. These two variables may optionally be set before building the library.

If you use the default Balau installation location (*${HOME}/usr*), both these variables will default to this. Otherwise, these variables must be set before building the library.

The exact method for specifying these CMake variables depends on whether you use the command line or an IDE. Only the command line technique is covered in the build steps here. For IDEs that support CMake, these are typically set from within the settings /preferences of the IDE. Refer to the specific IDE's documentation for information.

CMAKE_PREFIX_PATH

This CMake variable specifies a list of directories where dependencies may be found. More information is available on the CMake documentation here.

CMAKE_INSTALL_PREFIX

This CMake variable specifies the installation prefix into which the Balau library will be installed. More information is available on the CMake documentation here.

Environment variables

The Balau test application uses a number of environment variables in the unit tests, imported via the CMakeLists.txt file. These environment variables are optional. If they are not defined, the unit tests that require them will be disabled.

Refer to the CMakeLists.txt file for details on each environment variable if you wish to run the associated unit tests.

Building Balau

Building can be achieved either via the command line or from within an IDE that supports the CMake build system. Building and installing via the command line is covered in this document.

Linux

Open a command prompt and prepare the build with the following commands.

```
cd path/to/projects
git clone https://github.com/borasoftware/balau.git
cd path/to/balau/code
mkdir build-debug
cd build-debug
```

If you are using the default prefix path and install prefix, execute the following commands.

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make -j 4
make install
```

If you are using a non-default prefix path and/or a non-default install prefix, run the following commands, replacing *${DEPS}* with the installation locations of the dependencies you configured in the previous steps, and *${BALAU_PREFIX}* with the installation prefix where you wish to install the Balau library.

```
cmake -DCMAKE_PREFIX_PATH=${DEPS}             \
      -DCMAKE_INSTALL_PREFIX=${BALAU_PREFIX} \
      -DCMAKE_BUILD_TYPE=Debug               \
      ..

make -j 4
make install
```

The above set of commands:

- creates an out of source build directory *build-debug*;
- configures CMake;
- builds the library;
- installs the library to ${BALAU_PREFIX}.

If you wish to have a release build, set *CMAKE_BUILD_TYPE* to *Release* instead.

Windows

This section will be filled in when the Windows port has been completed.

Linking

In order to link to the Balau library, your *CMakeLists.txt* file needs to be modified with the Balau library and its dependencies.

These instructions have been written in order to use statically linked libraries. On some platforms, the ordering of the entries in the *CMakeLists.txt* file is important, in order that the linker may resolve the dependencies correctly.

The following CMake commands will ensure all libraries are found and linked correctly.

```
######################## BALAU ########################

find_package(Balau 2019.7.1 REQUIRED)
message(STATUS "Balau include dirs: ${Balau_INCLUDE_DIRS}")
message(STATUS "Balau library:      ${Balau_LIBRARY}")
include_directories(BEFORE ${Balau_INCLUDE_DIRS})
set(ALL_LIBS ${ALL_LIBS} ${Balau_LIBRARY})

################### BOOST LIBRARIES ###################

set(Boost_DETAILED_FAILURE_MSG ON)
set(Boost_USE_STATIC_LIBS ON)
set(Boost_USE_MULTITHREADED ON)
set(Boost_USE_STATIC_RUNTIME OFF)
# Add any other Boost libraries that you may require.
find_package(Boost 1.68.0 REQUIRED COMPONENTS thread chrono date_time files
include_directories(${Boost_INCLUDE_DIRS})
set(ALL_LIBS ${ALL_LIBS} ${Boost_LIBRARIES})

message(STATUS "Boost include dirs: ${Boost_INCLUDE_DIRS}")

######################### ICU #########################

find_package(ICU 60.2 REQUIRED COMPONENTS i18n uc data)
include_directories(${ICU_INCLUDE_DIRS})
set(ALL_LIBS ${ALL_LIBS} ${ICU_LIBRARIES})
message(STATUS "ICU include dirs: ${ICU_INCLUDE_DIRS}")
```

Contributing

Overview

Contributions to the Balau library are welcome. The easiest way to contribute is to create a fork of the library's repository and submit pull requests for new or enhanced features.

The Balau library has been conceived for two distinct aims:

- to provide a foundation for the development of Unicode aware C++ software applications that have a dependency injection based architecture, have complex logging requirements, and that will be developed with a test driven development methodology;
- to provide a simple and intuitive API for core C++ components and utilities.

Contributions to the injector, logging framework, and test runner are likely to be incremental improvements and bug fixes. Contributions of new components and improvements to other existing components and utilities are open to our imagination and creativity.

Planned features

The general aim of the library's development is reactive rather than proactive. If a useful feature for enterprise quality C++ application development is missing, convoluted, or lacking in features in the C++ standard library or Boost libraries, then the feature is a good candidate for development and inclusion in Balau.

If an existing feature in the standard library or Boost exists and is designed as low level, fine grained, or does not have a straight forward API, then it would be useful to develop a corresponding Balau feature with a simple and intuitive API, as a high level facade of the standard library or Boost feature.

Consequently, there is no complex plan of planned features to be added to the library. The current list of planned features can be seen on the planned features page.

License

Balau is licensed under the *Boost Software License - Version 1.0 - 2003*. All contributions will need to be licensed under the same license or under a license that allows relicensing under the Boost license.

Repository

The main repository is hosted at https://github.com/borasoftware/balau.

Guidelines

The following guidelines may help with contributions. In addition to avoiding defects, the aim is to maintain an easy to understand code base in order to facilitate with rapid application development.

General

- Classes of appreciable complexity are normally contained in their own header plus optional body file. Smaller classes should share a header with other similar classes / the complex class that uses them.
- Implementation specific classes are placed in files inside an *Impl* sub-folder next to the main class(es), and namespaced within an inner *Impl* sub-namespace. The injector and logging system are examples of this arrangement.
- Logging in Balau library components and functions should be restricted to code in body files only. This allows an incomplete declaration to be made for the *Logger* class in header files, thereby simplifying the include order in application code.
- Before creating a pull request, the code should be formatted to the Balau code style guidelines provided below.

Testing

- All anticipated uses cases of the new or enhanced feature should be tested by adding or revising the feature's test cases in the *src/test* folder.
- When a modification is made or new code is introduced and the tests are written, the test application should be run with *Valgrind* memcheck and helgrind tools, in order to verify that there are no memory and threading issues. If there are memory and/or threading issues, these should be debugged and resolved before raising a pull request.
- After completing the modification or new feature, the Balau test suite should be verified to pass in *release* mode in addition to *debug* mode.

Strings

- All features that use or manipulate *std::string* objects should be designed to function correctly with UTF-8 text, unless there is a valid reason not to do so (i.e. the data in the *std::string* is not UTF-8 text and/or is clearly processing pure bytes of information). Such exceptions should be documented as being so, otherwise it will be assumed that the bytes within a *std::string* object are UTF-8.

- The *wchar_t* character type and associated string and stream types must not be used anywhere in the library.
- UTF-8 character processing is supported by the ICU wrapper functions found within the *Balau::Character* namespace.
- When developing a utility function or component that manipulates UTF-8 strings, an equivalent UTF-32 version should normally be created alongside the UTF-8 version.
- When creating a function that accepts one or more const strings, string views should be used instead of std strings.
- Beware of the *std::string_view::data()* method. Do not use this method unless you are 100% sure null pointer character array termination is not required in the program logic that uses the data.
- Beware of returning *std::string_view* from methods. Returning *std::string_view* from a method should only be done when the lifetime of the referenced string data is guaranteed to outlive the string view being returned.

Const correctness

- All global variables, member variables, and local variables should be made *const* unless there is a reason to make them non-const.
- All member functions should be made *const* unless there is a reason to make them non-const.

Concurrency

- When developing code that relies on multi-threaded execution and inter-thread data sharing, the developed code should use standard C++ 11 memory ordering features, principally the atomic operations library. Mindful use of *std::memory_order* may improve performance on some platforms, but is usually not necessary, especially for the principal *x86-64* target platform.

Memory management

- Manual use of memory allocation via the *delete* operator should be avoided, except in exceptional circumstances where the code itself is acting as a pointer container that provides object lifetime management. Otherwise, the C++11 standard library pointer containers should be used for heap based objects.
- Use of C++11 move semantics should be preferred to copying. Copying should be limited to cases where the caller must retain ownership of the passed argument. When

copying is used, a pass by value and move approach should be used in preference to pass by reference and copy.

Templates

- Unnecessarily complex templated code should normally be avoided. Complex templated code should be used when the resulting solution is more elegant and/or the resulting public API is more simple than it would be if an alternative approach were taken AND quick start documentation for common use cases is provided. Our aim is to provide components and utilities for rapid enterprise quality application development. Development teams working on enterprise software applications do not normally have enough time for in-depth studies of the inner workings of a complexly templated component in order to fulfil a simple use case.
- Variadic functions and methods should always be implemented via C++11 parameter packs. Fold expressions may assist in the simplification of otherwise complex variadic tasks.

Macros

- The use of macros should be avoided, unless there is no other way of implementing a feature (examples include the file/line logging macros and the injector class macros).
- The use of conditional compilation macros should be avoided, unless there is no other way of implementing a feature. If some complex conditional compilation is required for resolving differences across platforms, put the code in platform specific headers and include them appropriately in the main source code file. The exception to this is the use of the *BALAU_DEBUG* macro, which is used to enable code in debug builds.

Documentation

- Public classes, functions, and methods should be documented with triple forward slash *///* documentation entries. Public items that are not documented will not be added to the API documentation. Each entry should have a single line description, a line break, then a more in-depth description. Parameters, exceptions, and return variables that are non-obvious should be documented with @tparam, @param, @throws, and @return entries. As Balau is a software library, users rely on the API documentation to use it, and consequently the API documentation is as essential as the code itself.
- Permanent block style /* */ code comments should never be used, as they complicate temporary block commenting/uncommenting.

- Protected and private classes, functions, and methods of any complexity should be documented with a brief, usually single line double forward slash *//* comment that indicates the goal of the entry. Triple forward slash */// *documentation entries should not be used unless the information for the item needs to appear in the API documentation.
- Code comments are generally unnecessary, unless a code fragment is unusually difficult to understand. In this case, a line or two of comments clarifying the goal of the fragment is useful.
- Author or version information should not be included in source code files. Detailed author information is available by examining the source code repository (git-blame).
- When a new class or feature is developed or an existing class or feature is enhanced, a corresponding BDML documentation page should be created or revised with the new or revised usage. The BDML documentation pages are found in the *src/doc /manual* folder. As the documentation is XML based, it can be written during development of the code and committed in the same change-sets as the code. Writing BDML documentation can be performed during compilation pauses. BDML documentation should follow the standard structure of *Overview*, *Quick start* (starting with the header(s) to include), optional detailed documentation sections, and optional *Design* section.

Code style

The aim of the code style used in the Balau library is maximum readability. The following sub-sections discuss aspects of the code style.

Indentation

Balau code style uses *smart tab* indenting. This allows developers to choose the tab size they desire in their source code editor, whilst maintaining correct alignment of vertically aligned items. Indentation size is thus not specified in this code style.

Files

Lines should be limited to 120 characters, when viewed with a tab size of 4 characters. Comment lines should normally be limited to approximately 80 characters.

Newlines in files must be *LF*, not *CRLF* or *CR*. Files should end with a newline. Whitespace should be removed at the ends of lines (configure the IDE to do it for you on saving).

Files should be named *"([A-Z][a-z]*)+.hpp"* for header files and *"([A-Z][a-z]*)+.cpp"* for body files.

Files should be grouped into a hierarchy of folders, the names of which are normally the same as the local namespace of the files contained within. The folder structure should normally follow the namespace structure.

Headers should use include barriers, the names of which exactly follow the names in the folder hierarchy in which the file is situated. The *#pragma once* definition should not be used. This will be revised when C++20 modules are standardized.

Includes in header files should be avoided when possible (use incomplete declarations). Otherwise, the full include path of non test header files should be used between < and > tokens.

```
// Example include in header file.
#include <Balau/Network/Http/Server/HttpWebApp.hpp>
```

Identifiers

Identifiers should be styled according to the following rules.

- Namespace, class, enum, and enum entry names should be upper camel case.
- Function, method, field names, and local variables should be lower camel case.
- Public macros (macros to be used in end application code) should be uppercase camel case.
- Private macros (macros to be used in other Balau macros) should be uppercase camel case and start with an underscore.
- Underscores should not be used apart from starting private macro identifiers and as a suffix for constructor parameters that set similarly named class fields in constructor initialisation lists.
- Identifiers other than local variables should be as short as possible whilst providing adequate information on their goals and avoiding abbreviations. It is fine to use long identifiers if there is no other way of providing sufficient information, but this should be the case for a minority of identifiers.
- Local variables, especially those used for temporary counters and the like, should be short, often a single letter.

Metadata naming conventions (type, kind) must not be used (use an IDE that indicates identifier semantics for you). Semantic apps identifiers should mostly be avoided, apart for template parameter identifiers.

Spacing

Spacing of source code tokens aims to maximise visual grouping of related tokens, whilst maintaining a compact representation. The spacing rules are specified as a subtractive list. All tokens should be surrounded by a space, with the exception of the following, which do not have space before and/or after them:

- after '(', '[', and before ']', ')';
- after '<' and before '>' when these characters are template header tokens;
- before '(' and after ')' when these characters are function call parameter list parentheses;
- after '::';
- before '::', unless the character pair references the global scope;
- between *public*, *protected*, and *private* tokens and their associated ':';
- between 'case' and its associated ':' in a switch statement;
- before ',' and ';'
- after '*' and '&' when these characters are de-reference and address-of tokens;
- after '+' and '-' when these characters are unary operators.

Braces

Opening '{' braces are placed on the same line as the associated statement.

Single line code blocks should not be used.

Single line code blocks must use braces.

```
// Single line code block in if statement.
if (i < 0) {
    foo();
}
```

Case blocks within switch statements should also use braces.

```
// Switch statement case blocks.
switch (type) {
    case Type::Simple: {
        foo();
        break;
    }

    case Type::Composite: {
        foo2();
        break;
    }

    default: {
        throwError();
        break;
    }
}
```

Horizontal/vertical

This principal maximises the readability of parameter lists, enum entries, argument lists, and other delimited lists found in the source code.

The general idea is that it is easiest to read a delimited list when it is presented either in a single line or as a vertically aligned list. The choice of the two approaches is dictated by the length of the line when the delimited list is presented on a single line. If a single line fits within the 120 character limit, a single line is chosen. Otherwise, a vertically aligned list is chosen.

The following extracts illustrate this approach on method parameter lists.

```
// From HttpWebApp class.

public: virtual void handleGetRequest(HttpSession & session, const Request
```

In this example, the method's header fits on a single line without overrunning the 120 character limit.

```
// From HttpServer class.

public: HttpServer(std::shared_ptr<Injector> injector,
                   const std::string & serverIdentification,
                   const TCP::endpoint & endpoint,
                   std::string threadNamePrefix_,
                   size_t workerCount_,
                   std::shared_ptr<HttpWebApp> httpHandler,
                   std::shared_ptr<WsWebApp> wsHandler,
                   std::shared_ptr<MimeTypes> mimeTypes = MimeTypes::defau
```

In this example, the method's header would overrun the 120 character limit if it were presented on a single line, so the parameter list is arranged vertically aligned.

Delimiters

With the exception of commas in function and method parameter lists (as illustrated in the previous code example), the delimiters in a vertically aligned, delimiter separated list are each considered to belong to the following item in the list.

For example, the commas in the vertically delimited superclass call in the following code extract lead the arguments.

```
protected: MultiProcessTestRunnerExecutor(CompositeWriter & writer_,
                                          bool useNamespaces_,
                                          GroupedTestCaseMap & testCases,
                                          unsigned int concurrencyLevel_)
    : TestRunnerExecutor(
          std::unique_ptr<TestResultQueue>(new MultiProcessTestResultQueue
        , writer_
        , useNamespaces_
        , testCases
        , true
    )
    , concurrencyLevel(concurrencyLevel_)
    , sharedMemoryNamePrefix(createSharedMemoryNamePrefix()) {}
```

This code extract also illustrates comma delimited field initialisation.

Closing brackets

When a vertically aligned list is used within opening and closing brackets/parentheses, the closing bracket/parenthesis is placed on a newline. An example of this is shown in the previous code extract. The following is an example with two levels.

```
tests.emplace_back(
    FlattenedTestCase(
          testIndex
        , executionModels
        , preText
        , ""
        , testCase.name
        , testCase.group
        , std::move(testCase.method)
    )
);
```

Visibility prefixes

The visibility prefixes are the *public*, *protected*, and *private* tokens used in class declarations.

In Balau sources, class declarations use explicit visibility prefixes on all declaration items (fields, methods, inner class/enum declarations). This provides physically collocated visibility information for all items, avoiding the need to search upwards in the source code for the visibility of a class item and declaring that the item is part of a class declaration.

Known issues

The following issues are outstanding.

Issue	Description
Possible memory leak in OpenSSL 1.1	Valgrind is reporting a memory leak in Boost Asio *SSL::context*. The call stack ends with a malloc call initiated by the OpenSSL call *ERR_get_state* (tested OpenSSL version 1.1). However, since version 1.1, OpenSSL is reported to clean up all thread local allocations itself.

Planned features

The following features and enhancements are currently planned.

Platforms

- Port to Windows 7/10.

General

File system

Migrate from *<boost/filesystem.hpp>* to the standard library replacement *<filesystem>* once there is widespread support.

Testing

Improve test coverage and increase use case tests on each component (explicitly test functionality that is used elsewhere but not directly tested).

Error reporting

Improve exception messages and information provided at exception throwing sites.

Components

Application

Injector

- Investigate the viability of implementing automatic constructor selection, allowing injectable implementation classes to be written without the need for an injector macro.
- Improve dependency tree cycle error reporting.
- Improve dependency tree pretty printing.
- Improve the injector unit tests.
- Implement parallel construction of eager singletons, based on the dependency tree created during the validation phase of injector construction.
- *[non functional]* Consider the possibility of compacting the injector macro implementations via some macro twiddling.

Environment

- Add documentation describing how to construct custom type specification hierarchies from existing type specification files.
- Consider the implementation of composite property arrays via multiple specification of the same composite property.
- Create a web application that serves environment configuration property specifications, given a list of type specification source files.

Command line parser

- Improve the help text printer.

Concurrent

- Determine how to add processing forking simulation for the Windows 7/10 platform and add the functionality to the *Fork* class.
- Determine which other useful (high level API) concurrent data structures are not available from the standard library / Boost libraries and implement.

Container

- Add more search algorithms to *ObjectTrie*.
- Add an efficient single process blocking queue implementation.
- Determine which other useful (high level API) containers are not available from the standard library / Boost libraries and implement.

Lang

- Extend the parser utilities classes to support incremental parsing.

Logging

- Create a threaded writer in order to offload writing to another thread.

Network

- *[non functional]* Consider whether to migrate away from libcurl (i.e. write C++ SMTP handling and other handlers).

- Add features to the HTTP and HTTPS clients (start with chunked transfer and asynchronous API, redirects).
- Make HTTP server keep-alive configurable.
- Add TLS support to the HTTP server.
- Improve/add other features to the HTTP server (chunked transfer, etc.).
- Consider which other web application handlers would be useful.
- Analyse the performance of the HTTP server and determine how to improve performance.
- Create a more efficient regular expression algorithm/data structure for the redirecting HTTP web application.

Resource

- Add more resource types.
- *[non functional]* Improve the resource documentation in the developer manual.

System

- Add more features to the clock API and the system clock implementation.

Testing

- Add test case serialisation via dependency keys.
- Integrate with common C++ unit test running frameworks.

Type

- Clean up / improve templated (typename AllocatorT) versions of the toString, toString16, and toString32 functions.
- Migrate fromString numeric conversions from c-string type *strtol* etc. to C++17 *from_chars* functions. This will avoid the need to allocate strings from string views.

Util

Compression

GZip

- Add functions that compress to *std::vector<char>*.

- Add functions that compress to *std::ostream*.
- Add functions that uncompress from a *std::vector<char>*.
- Add functions that uncompress from a *std::istream*.

Zipper/Unzipper

- Make the zip classes more feature complete.

Reporting bugs

The Balau library is in active development. Although care has been taken to define test cases for a variety of use cases for each component, it is possible that an untested use case may have a defect.

The proposed procedure for reporting and fixing defects is as follows.

1. Create one or more Balau test cases for the defect. All defect reports should include at least one test case that reproduces the defect. One option to achieve this is to create your own fork of the library and add your test case to your repository.
2. Either: report the defect on the main repository issue page here;
3. Or: fix the defect in your own fork and submit a pull request to merge it to the main repository.