Message bus: simple event framework

Goal : Decouple when a message or event is sent from when it is processed.

Files:

- eventApi.h — Public api
- containerWrapper.h — Incomplete attempt on wrapping stl thread safe
- eventFrameWork.h — Implementation
- eventFramework.cpp — Implementation

- Readme.pdf, Readme.txt
- main.cpp — runner
- testBus.h — gtests for components
- testComponents.h — gtests for the bus/api

Build instructions:

I used some constructs from C++14 standard.
Also includes gtest libraries, no other external dependencies.

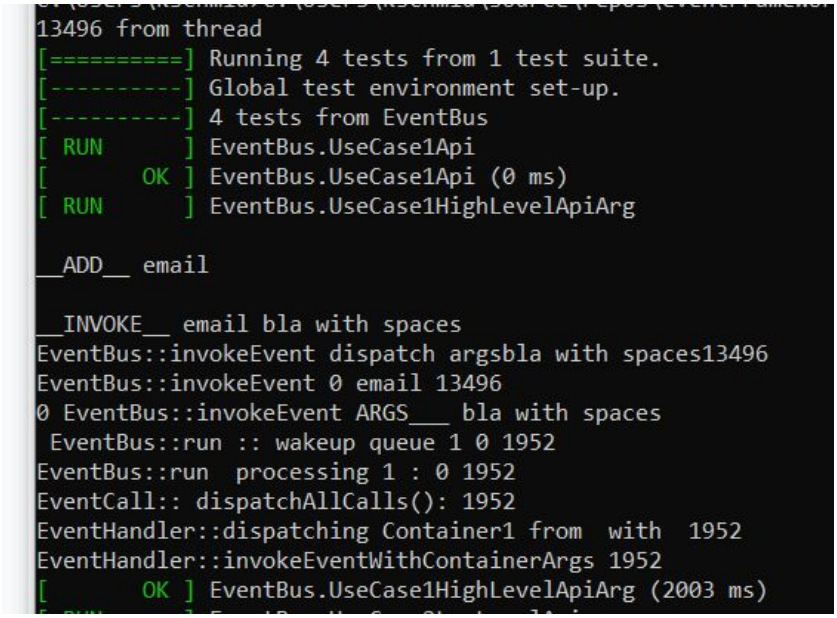
Running

Please use api documented in eventApi.h. More examples in testBus.h
Minimal example:

```
#include "eventApi.h"

const std::string iFunctionName1("email");
std::function<void(std::string)> executeMe = ([=](std::string s)
                -> void {  std::this_thread::sleep_for(std::chrono::milliseconds(100));
std::cout << "\nexecuteMeParam test2 " << s << std::endl; });
add(iFunctionName1, executeMe);
invoke(iFunctionName1);
```

For a lot of debug output it is possible to use m_verbose mode on the classes.



Assumptions and design decisions

The system has queue of callable objects storing lambda functions.
When a new eventCall is added to queue there is a signa for the thread to continue execution of the event loop. This function exits when stop() is called. After stopping use reset() member function to enable the state for it to execute again.

Important classes

ArgumentContainer mArgument<T>		**EventBase** Polymorphic Name Verbose EventType isValid

Eventbus		**Event <T>**
Runnigstate **map<EventHandler *> 0…* //blocked** **queque<EventCall *> 0…***		*ResultState* *Runnigstate* **Callback<T>**
reset() add() invoke() run() stop()		invoke() const
EventCall/Job		**EventHandler**
Runnigstate *ResultState* *Timestamp* Eventhandler ArgumentContainerBase *		Runnigstate ResultState vector<EventBase *> 1…*//
invokeAll(argument T) const		addEvent<T> dispatchAll

Storing events:

- Callables are stored in a Event. Events are grouped in a Eventhandler
- EventCalls have an Eventhandler, Eventhandler can have multiple objects
- Each of them have states and setup Arguments are stored in their own container
- EventHandlers are stored in map for efficient access by string
- Tries to store as much additional information for the callback as available.
  Because of limited Introspection
    - I think there may not be an easy way to avoid duplicates I just append all callbacks (even the invalid ones because of blocking, see below)
    - I don't think we can know the expected argument list

Invoking an event:

My interpretation from the given example is that callbacks by the same name don't override each other, rather they are added (unless they are found invalid or blocked).
Arguments go into a custom container intended to hide typing. I had to abandon making anything but strings and void work because of time constraints.
There isn't enough information in invoke() to distinguish between callbacks by the same name so all valid callback by the name are called in the order they are added.

The EventCall saves the arguments for a call and will call the EventHandler who invokes the callsbacks on the Events.

Canceling an event:

The description is ambiguous and the example leaves room for interpretations. In real life that would need to be clarified before coding, here I am just gong simplest interpretation that makes sense to me in the context. Setting an invalid callback object disables that particular call in the EventHandler container by adding a “blocked” state. The callback will not be invoked while the state is set.

Multithreading:

All standard library containers are not thread safe c14 I tried wrapping stl containers I am using.
It's a can of worms as expected.

TODOs/ More Features to add

- Logging, Docstrings, Getter setter, Move code out of the header
- Failure handling like Retry, timeouts, handling failed jobs, Keeping tasks around until they are finished
- Typing function/args, more reading on that for me
- Way more testing, bad cases
- I also put type deduction and variadic templates on my reading list