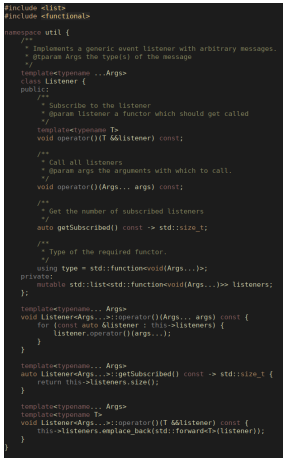
Team-10

Server

May 7, 2019

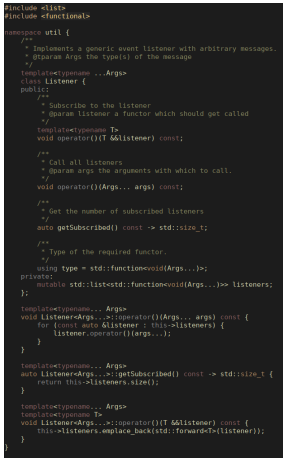
Überblick



```cpp
#include <list>
#include <functional>

namespace util {
    /**
     * Implements a generic event listener with arbitrary messages.
     * @tparam Args the type(s) of the message
     */
    template<typename ...Args>
    class Listener {
    public:
        /**
         * Subscribe to the listener
         * @param listener a functor which should get called
         */
        template<typename T>
        void operator()(T &&listener) const;

        /**
         * Call all listeners
         * @param args the arguments with which to call.
         */
        void operator()(Args... args) const;

        /**
         * Get the number of subscribed listeners
         */
        auto getSubscribed() const -> std::size_t;

        /**
         * Type of the required functor.
         */
        using type = std::function<void(Args...)>;
    private:
        mutable std::list<std::function<void(Args...)>> listeners;
    };

    template<typename... Args>
    void Listener<Args...>::operator()(Args... args) const {
        for (const auto &listener : this->listeners) {
            listener.operator()(args...);
        }
    }

    template<typename... Args>
    auto Listener<Args...>::getSubscribed() const -> std::size_t {
        return this->listeners.size();
    }

    template<typename... Args>
    template<typename T>
    void Listener<Args...>::operator()(T &&listener) const {
        this->listeners.emplace_back(std::forward<T>(listener));
    }
}
```

- Vollständig in C++17 geschrieben

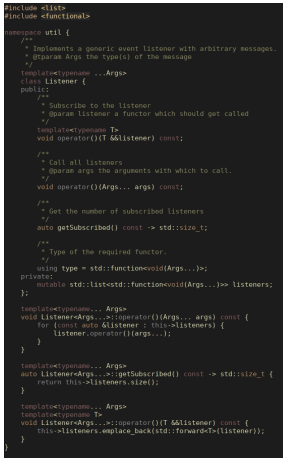
Überblick



```cpp
#include <list>
#include <functional>

namespace util {
    /**
     * Implements a generic event listener with arbitrary messages.
     * @tparam Args the type(s) of the message
     */
    template<typename ...Args>
    class Listener {
    public:
        /**
         * Subscribe to the listener
         * @param listener a functor which should get called
         */
        template<typename T>
        void operator()(T &&listener) const;

        /**
         * Call all listeners
         * @param args the arguments with which to call.
         */
        void operator()(Args... args) const;

        /**
         * Get the number of subscribed listeners
         */
        auto getSubscribed() const -> std::size_t;

        /**
         * Type of the required functor.
         */
        using type = std::function<void(Args...)>;
    private:
        mutable std::list<std::function<void(Args...)>> listeners;
    };

    template<typename... Args>
    void Listener<Args...>::operator()(Args... args) const {
        for (const auto &listener : this->listeners) {
            listener.operator()(args...);
        }
    }

    template<typename... Args>
    auto Listener<Args...>::getSubscribed() const -> std::size_t {
        return this->listeners.size();
    }

    template<typename... Args>
    template<typename T>
    void Listener<Args...>::operator()(T &&listener) const {
        this->listeners.emplace_back(std::forward<T>(listener));
    }
}
```

- Vollständig in C++17 geschrieben
- Modularer Aufbau (Trennung in Network-, Messages-, GameLogic- und Serverkomponente)

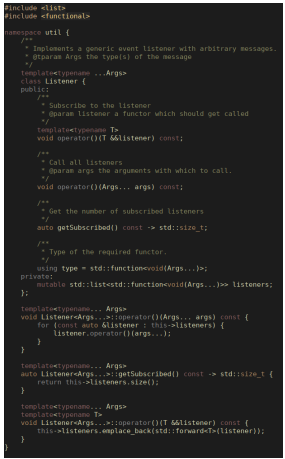
Überblick



```
#include <list>
#include <functional>

namespace util {
    /**
     * Implements a generic event listener with arbitrary messages.
     * @tparam Args the type(s) of the message
     */
    template<typename ...Args>
    class Listener {
    public:
        /**
         * Subscribe to the listener
         * @param listener a functor which should get called
         */
        template<typename T>
        void operator()(T &&listener) const;

        /**
         * Call all listeners
         * @param args the arguments with which to call.
         */
        void operator()(Args... args) const;

        /**
         * Get the number of subscribed listeners
         */
        auto getSubscribed() const -> std::size_t;

        /**
         * Type of the required functor.
         */
        using type = std::function<void(Args...)>;
    private:
        mutable std::list<std::function<void(Args...)>> listeners;
    };

    template<typename... Args>
    void Listener<Args...>::operator()(Args... args) const {
        for (const auto &listener : this->listeners) {
            listener.operator()(args...);
        }
    }

    template<typename... Args>
    auto Listener<Args...>::getSubscribed() const -> std::size_t {
        return this->listeners.size();
    }

    template<typename... Args>
    template<typename T>
    void Listener<Args...>::operator()(T &&listener) const {
        this->listeners.emplace_back(std::forward<T>(listener));
    }
}
```

- Vollständig in C++17 geschrieben
- Modularer Aufbau (Trennung in Network-, Messages-, GameLogic- und Serverkomponente)
- MVC

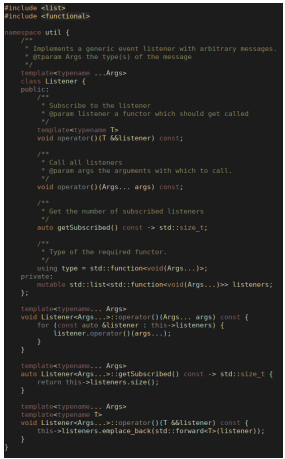
Überblick



```
#include <list>
#include <functional>

namespace util {
    /**
     * Implements a generic event listener with arbitrary messages.
     * @tparam Args the type(s) of the message
     */
    template<typename ...Args>
    class Listener {
    public:
        /**
         * Subscribe to the listener
         * @param listener a functor which should get called
         */
        template<typename T>
        void operator()(T &&listener) const;

        /**
         * Call all listeners
         * @param args the arguments with which to call.
         */
        void operator()(Args... args) const;

        /**
         * Get the number of subscribed listeners
         */
        auto getSubscribed() const -> std::size_t;

        /**
         * Type of the required functor.
         */
        using type = std::function<void(Args...)>;
    private:
        mutable std::list<std::function<void(Args...)>> listeners;
    };

    template<typename... Args>
    void Listener<Args...>::operator()(Args... args) const {
        for (const auto &listener : this->listeners) {
            listener.operator()(args...);
        }
    }

    template<typename... Args>
    auto Listener<Args...>::getSubscribed() const -> std::size_t {
        return this->listeners.size();
    }

    template<typename... Args>
    template<typename T>
    void Listener<Args...>::operator()(T &&listener) const {
        this->listeners.emplace_back(std::forward<T>(listener));
    }
}
```

- Vollständig in C++17 geschrieben
- Modularer Aufbau (Trennung in Network-, Messages-, GameLogic- und Serverkomponente)
- MVC
- Replay implementiert

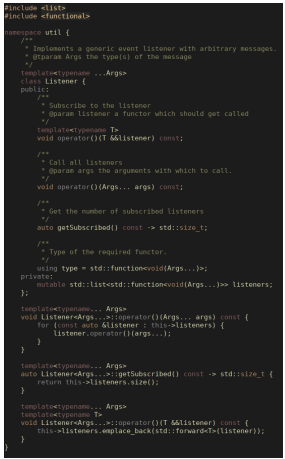
Überblick



```
#include <list>
#include <functional>

namespace util {
    /**
     * Implements a generic event listener with arbitrary messages.
     * @tparam Args the type(s) of the message
     */
    template<typename ...Args>
    class Listener {
    public:
        /**
         * Subscribe to the listener
         * @param listener a functor which should get called
         */
        template<typename T>
        void operator()(T &&listener) const;

        /**
         * Call all listeners
         * @param args the arguments with which to call.
         */
        void operator()(Args... args) const;

        /**
         * Get the number of subscribed listeners
         */
        auto getSubscribed() const -> std::size_t;

        /**
         * Type of the required functor.
         */
        using type = std::function<void(Args...)>;
    private:
        mutable std::list<std::function<void(Args...)>> listeners;
    };

    template<typename... Args>
    void Listener<Args...>::operator()(Args... args) const {
        for (const auto &listener : this->listeners) {
            listener.operator()(args...);
        }
    }

    template<typename... Args>
    auto Listener<Args...>::getSubscribed() const -> std::size_t {
        return this->listeners.size();
    }

    template<typename... Args>
    template<typename T>
    void Listener<Args...>::operator()(T &&listener) const {
        this->listeners.emplace_back(std::forward<T>(listener));
    }
}
```

- Vollständig in C++17 geschrieben
- Modularer Aufbau (Trennung in Network-, Messages-, GameLogic- und Serverkomponente)
- MVC
- Replay implementiert
- Alle Mods implementiert

Überblick



```
#include <list>
#include <functional>

namespace util {
    /**
     * Implements a generic event listener with arbitrary messages.
     * @tparam Args the type(s) of the message
     */
    template<typename ...Args>
    class Listener {
    public:
        /**
         * Subscribe to the listener
         * @param listener a functor which should get called
         */
        template<typename T>
        void operator()(T &&listener) const;

        /**
         * Call all listeners
         * @param args the arguments with which to call.
         */
        void operator()(Args... args) const;

        /**
         * Get the number of subscribed listeners
         */
        auto getSubscribed() const -> std::size_t;

        /**
         * Type of the required functor.
         */
        using type = std::function<void(Args...)>;
    private:
        mutable std::list<std::function<void(Args...)>> listeners;
    };

    template<typename... Args>
    void Listener<Args...>::operator()(Args... args) const {
        for (const auto &listener : this->listeners) {
            listener.operator()(args...);
        }
    }

    template<typename... Args>
    auto Listener<Args...>::getSubscribed() const -> std::size_t {
        return this->listeners.size();
    }

    template<typename... Args>
    template<typename T>
    void Listener<Args...>::operator()(T &&listener) const {
        this->listeners.emplace_back(std::forward<T>(listener));
    }
}
```

- Vollständig in C++17 geschrieben
- Modularer Aufbau (Trennung in Network-, Messages-, GameLogic- und Serverkomponente)
- MVC
- Replay implementiert
- Alle Mods implementiert
- Mehrere zeitgleiche Spiele

Krasse Features

- Ausführliches README mit Anleitung zum Installieren und Nutzen

Server

Server component for the Quidditch Game.

Getting started

You can choose between using Docker or manually installing all dependencies. Docker is the preferred method as it already installs the toolchain and all dependencies.

Docker

In the root directory of the project build the docker image ("server" is the name of the container, this can be replaced by a different name):

```
docker build -t server .
```

Now start the container, you need to map the internal port (8080 by default, to some external port 80 in this case) and map the external file (match.json) to an internal file:

```
docker run -v ${pwd}/match.json:match.json -p 80:8080 server ./Server -m /match.json -p 8080
```

That's it you should now have a running docker instance.

Manually installing the Server

If you need to debug the server it can be easier to do this outside of docker.

Prerequisites

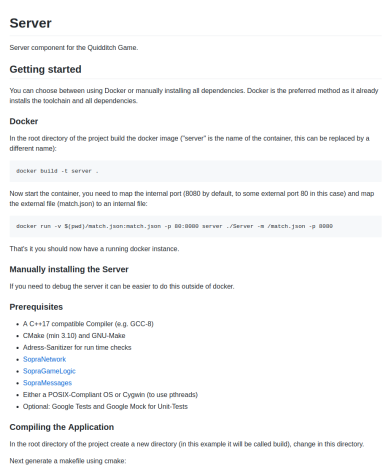
- A C++17 compatible Compiler (e.g. GCC-8)
- CMake (min 3.10) and GNU-Make
- Adress-Sanitizer for run time checks
- SopraNetwork
- SopraGameLogic
- SopraMessages
- Either a POSIX-Compliant OS or Cygwin (to use pthreads)
- Optional: Google Tests and Google Mock for Unit-Tests

Compiling the Application

In the root directory of the project create a new directory (in this example it will be called build), change in this directory.

Next generate a makefile using cmake:

Krasse Features



Server

Server component for the Quidditch Game.

Getting started

You can choose between using Docker or manually installing all dependencies. Docker is the preferred method as it already installs the toolchain and all dependencies.

Docker

In the root directory of the project build the docker image ("server" is the name of the container, this can be replaced by a different name):

```
docker build -t server .
```

Now start the container, you need to map the internal port (8080 by default, to some external port 80 in this case) and map the external file (match.json) to an internal file:

```
docker run -v ${pwd}/match.json:match.json -p 80:8080 server ./Server -m /match.json -p 8080
```

That's it you should now have a running docker instance.

Manually installing the Server

If you need to debug the server it can be easier to do this outside of docker.

Prerequisites

- A C++17 compatible Compiler (e.g. GCC-8)
- CMake (min 3.10) and GNU-Make
- Adress-Sanitizer for run time checks
- SopraNetwork
- SopraGameLogic
- SopraMessages
- Either a POSIX-Compliant OS or Cygwin (to use pthreads)
- Optional: Google Tests and Google Mock for Unit-Tests

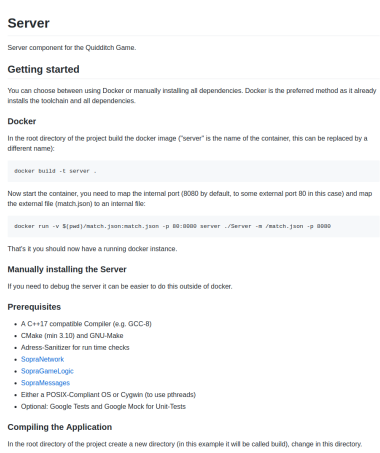
Compiling the Application

In the root directory of the project create a new directory (in this example it will be called build), change in this directory.

Next generate a makefile using cmake:

- Ausführliches README mit Anleitung zum Installieren und Nutzen
- Mit Doxygen dokumentiert

Krasse Features



Server

Server component for the Quidditch Game.

Getting started

You can choose between using Docker or manually installing all dependencies. Docker is the preferred method as it already installs the toolchain and all dependencies.

Docker

In the root directory of the project build the docker image ("server" is the name of the container, this can be replaced by a different name):

```
docker build -t server .
```

Now start the container, you need to map the internal port (8080 by default, to some external port 80 in this case) and map the external file (match.json) to an internal file:

```
docker run -v ${pwd}/match.json:match.json -p 80:8080 server ./Server -m /match.json -p 8080
```

That's it you should now have a running docker instance.

Manually installing the Server

If you need to debug the server it can be easier to do this outside of docker.

Prerequisites

- A C++17 compatible Compiler (e.g. GCC-8)
- CMake (min 3.10) and GNU-Make
- Adress-Sanitizer for run time checks
- SopraNetwork
- SopraGameLogic
- SopraMessages
- Either a POSIX-Compliant OS or Cygwin (to use pthreads)
- Optional: Google Tests and Google Mock for Unit-Tests

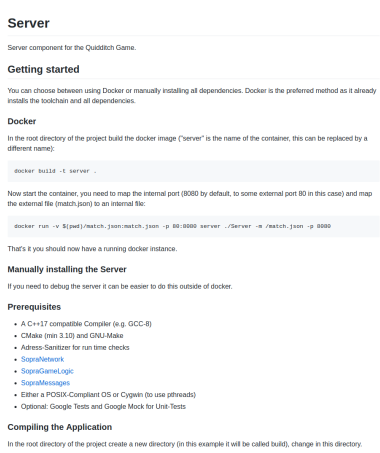
Compiling the Application

In the root directory of the project create a new directory (in this example it will be called build), change in this directory.

Next generate a makefile using cmake:

- Ausführliches README mit Anleitung zum Installieren und Nutzen
- Mit Doxygen dokumentiert
- 263 Unit Tests

Krasse Features



Server

Server component for the Quidditch Game.

Getting started

You can choose between using Docker or manually installing all dependencies. Docker is the preferred method as it already installs the toolchain and all dependencies.

Docker

In the root directory of the project build the docker image ("server" is the name of the container, this can be replaced by a different name):

```
docker build -t server .
```

Now start the container, you need to map the internal port (8080 by default, to some external port 80 in this case) and map the external file (match.json) to an internal file:

```
docker run -v ${pwd}/match.json:match.json -p 80:8080 server ./Server -m /match.json -p 8080
```

That's it you should now have a running docker instance.

Manually installing the Server

If you need to debug the server it can be easier to do this outside of docker.

Prerequisites

- A C++17 compatible Compiler (e.g. GCC-8)
- CMake (min 3.10) and GNU-Make
- Adress-Sanitizer for run time checks
- SopraNetwork
- SopraGameLogic
- SopraMessages
- Either a POSIX-Compliant OS or Cygwin (to use pthreads)
- Optional: Google Tests and Google Mock for Unit-Tests

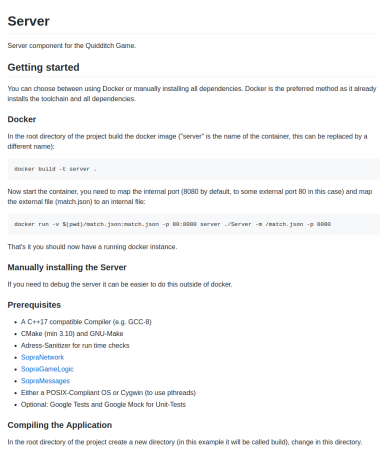
Compiling the Application

In the root directory of the project create a new directory (in this example it will be called build), change in this directory.

Next generate a makefile using cmake:

- Ausführliches README mit Anleitung zum Installieren und Nutzen
- Mit Doxygen dokumentiert
- 263 Unit Tests
- Statische und Dynamische Analyse (Clang-Tidy und Address-Sanitizer)

Krasse Features



Server

Server component for the Quidditch Game.

Getting started

You can choose between using Docker or manually installing all dependencies. Docker is the preferred method as it already installs the toolchain and all dependencies.

Docker

In the root directory of the project build the docker image ("server" is the name of the container, this can be replaced by a different name):

```
docker build -t server .
```

Now start the container, you need to map the internal port (8080 by default, to some external port 80 in this case) and map the external file (match.json) to an internal file:

```
docker run -v ${pwd}/match.json:match.json -p 80:8080 server ./Server -m /match.json -p 8080
```

That's it you should now have a running docker instance.

Manually installing the Server

If you need to debug the server it can be easier to do this outside of docker.

Prerequisites

- A C++17 compatible Compiler (e.g. GCC-8)
- CMake (min 3.10) and GNU-Make
- Adress-Sanitizer for run time checks
- SopraNetwork
- SopraGameLogic
- SopraMessages
- Either a POSIX-Compliant OS or Cygwin (to use pthreads)
- Optional: Google Tests and Google Mock for Unit-Tests

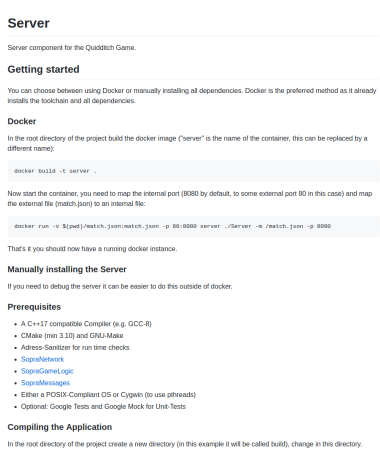
Compiling the Application

In the root directory of the project create a new directory (in this example it will be called build), change in this directory.

Next generate a makefile using cmake:

- Ausführliches README mit Anleitung zum Installieren und Nutzen
- Mit Doxygen dokumentiert
- 263 Unit Tests
- Statische und Dynamische Analyse (Clang-Tidy und Address-Sanitizer)
- Keine Warnings, komplette Entwicklung mit Warnings als Error (*-Werror*)

Krasse Features



Server

Server component for the Quidditch Game.

Getting started

You can choose between using Docker or manually installing all dependencies. Docker is the preferred method as it already installs the toolchain and all dependencies.

Docker

In the root directory of the project build the docker image ("server" is the name of the container, this can be replaced by a different name):

```
docker build -t server .
```

Now start the container, you need to map the internal port (8080 by default, to some external port 80 in this case) and map the external file (match.json) to an internal file:

```
docker run -v ${pwd}/match.json:match.json -p 80:8080 server ./Server -m /match.json -p 8080
```

That's it you should now have a running docker instance.

Manually installing the Server

If you need to debug the server it can be easier to do this outside of docker.

Prerequisites

- A C++17 compatible Compiler (e.g. GCC-8)
- CMake (min 3.10) and GNU-Make
- Adress-Sanitizer for run time checks
- SopraNetwork
- SopraGameLogic
- SopraMessages
- Either a POSIX-Compliant OS or Cygwin (to use pthreads)
- Optional: Google Tests and Google Mock for Unit-Tests

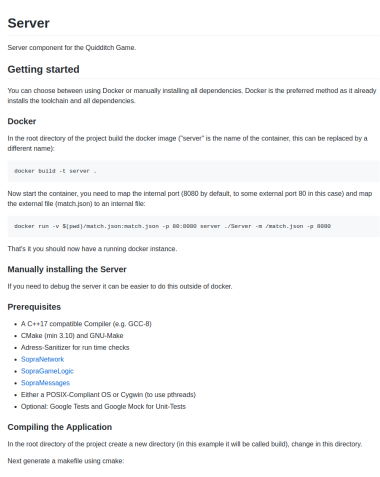
Compiling the Application

In the root directory of the project create a new directory (in this example it will be called build), change in this directory.

Next generate a makefile using cmake:

- Ausführliches README mit Anleitung zum Installieren und Nutzen
- Mit Doxygen dokumentiert
- 263 Unit Tests
- Statische und Dynamische Analyse (Clang-Tidy und Address-Sanitizer)
- Keine Warnings, komplette Entwicklung mit Warnings als Error (*-Werror*)
- Fertiges Docker-Image

Krasse Features



Server

Server component for the Quidditch Game.

Getting started

You can choose between using Docker or manually installing all dependencies. Docker is the preferred method as it already installs the toolchain and all dependencies.

Docker

In the root directory of the project build the docker image ("server" is the name of the container, this can be replaced by a different name):

```
docker build -t server .
```

Now start the container, you need to map the internal port (8080 by default, to some external port 80 in this case) and map the external file (match.json) to an internal file:

```
docker run -v ${pwd}/match.json:match.json -p 80:8080 server ./Server -m /match.json -p 8080
```

That's it you should now have a running docker instance.

Manually installing the Server

If you need to debug the server it can be easier to do this outside of docker.

Prerequisites

- A C++17 compatible Compiler (e.g. GCC-8)
- CMake (min 3.10) and GNU-Make
- Adress-Sanitizer for run time checks
- SopraNetwork
- SopraGameLogic
- SopraMessages
- Either a POSIX-Compliant OS or Cygwin (to use pthreads)
- Optional: Google Tests and Google Mock for Unit-Tests

Compiling the Application

In the root directory of the project create a new directory (in this example it will be called build), change in this directory.

Next generate a makefile using cmake:

- Ausführliches README mit Anleitung zum Installieren und Nutzen
- Mit Doxygen dokumentiert
- 263 Unit Tests
- Statische und Dynamische Analyse (Clang-Tidy und Address-Sanitizer)
- Keine Warnings, komplette Entwicklung mit Warnings als Error (*-Werror*)
- Fertiges Docker-Image
- CI (Testen des Docker-Images, Unittests (mehrfache Wiederholung in unterschiedlicher Reihenfolge))