Instrumenting the Stack

Strategies for End-to-end Sanitizer Adoption

DAMIEN BUHL

Cppcon
The C++ Conference

2025
September 13 - 19

EngFlow

→ Co-Founded by the lead developer of Bazel, Google's open-source build system

→ Acquired tipi.build the team behind CMake RE

→ Co-founded by Boost Contributors, CMake lovers and Standard C++ ISO members

Texas-based company acquires tipi.build

13.03.2025

XPRENEURS INCUBATOR · ALCHEMIST ACCELERATOR · andreessen. horowitz · AICPA SOC · aws activate · Google Cloud Partner

CMake RE

Drop-in for CMake

```
cmake \
  -G Ninja -S . -B build/ \
  -DCMAKE_TOOLCHAIN_FILE=toolchains/linux-ubuntu24.04-clang20.cmake \
  -DCMAKE_BUILD_TYPE=Release

cmake --build build/
```

CMake RE

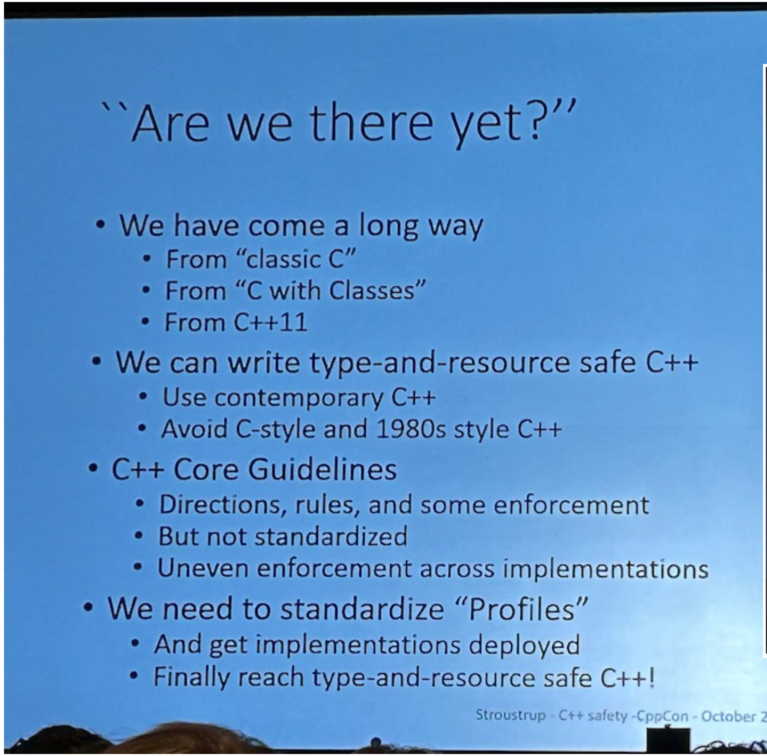
Drop-in for CMake

```
cmake \
  -G Ninja -S . -B build/ \
  -DCMAKE_TOOLCHAIN_FILE=toolchains/linux-ubuntu24.04-clang20.cmake \
  -DCMAKE_BUILD_TYPE=Release

cmake --build build/
```

CMake with Remote, Distributed, Containerized Execution and Caching

```
cmake-re [--distributed] [--remote|--host] \
  -G Ninja -S . -B build/ \
  -DCMAKE_TOOLCHAIN_FILE=toolchains/linux-ubuntu24.04-clang20.cmake \
  -DCMAKE_BUILD_TYPE=Release

cmake-re [--jobs 1000+] --build build/
```

CppCon 2023 : Delivering Safe C++



``Are we there yet?''

- We have come a long way
  - From "classic C"
  - From "C with Classes"
  - From C++11
- We can write type-and-resource safe C++
  - Use contemporary C++
  - Avoid C-style and 1980s style C++
- C++ Core Guidelines
  - Directions, rules, and some enforcement
  - But not standardized
  - Uneven enforcement across implementations
- We need to standardize "Profiles"
  - And get implementations deployed
  - Finally reach type-and-resource safe C++!

Stroustrup - C++ safety -CppCon - October 2023

Delivering Safe C++ - Bjarne Stroustrup - CppCon 2023

Are we safe yet ?

→ Not fully yet, but in practice it’s doable.

→ Good tooling exists, it is just waiting for teams to use it.

→ Requires more build types and test pipelines.

Practical Ways of getting Safety in C++

Productivity vs Safety

→ Setting up tools

→ Good test code coverage

→ Requires more build types, means longer iterations in getting code merged

Safety Toolbox

Static Analysis : clang-tidy 1

Using Sanitizers : MSan, ASan + LSan + UBSan, TSan 2

TSA (Thread Safety Analysis) 3

clang-tidy : Static Analysis

Default set of checks

- clang-analyzer-*: Clang Static Analyzer detect potential bugs and vulnerabilities
- cppcoreguidelines-*: Enforces the C++ Core Guidelines
- bugprone-*: Identify common programming mistakes that often lead to bugs

→ https://clang.llvm.org/extra/clang-tidy/checks/list.html

→ Parses code twice

- Once for clang-tidy
- Once for compilation

→ Checks can be silenced in C++ code with `// NOLINT`

clang-tidy : Static Analysis

```cmake
# Enable clang-tidy checks
set(CMAKE_CXX_CLANG_TIDY clang-tidy)
set(CMAKE_C_CLANG_TIDY clang-tidy)
```

MSan : MemorySanitizer

Detector of Uninitialized Memory

→ **Shadow Memory**
1 corresponding bit for each application’s memory bits.
Mark Poisoned (1) on allocation, unpoisoned (0) on initialization.

→ **Compiler Instrumentation on**

- Memory allocation
- Initialization (e.g. assignments, memset, memcpy)
- Destructors to catch use-after-destruction
- Memory Read, dereference, function call

→ **Doubles the execution time**

- Faster than Valgrind which slows down execution by a factor 20
- Linux only

MSan : How to get full-instrumentation

```
RUN mkdir -p /llvm-project && \
  git clone --branch llvmorg-20.1.2 --depth 1 https://github.com/llvm/llvm-project.git /llvm-project && \
  cmake \
  -DCMAKE_BUILD_TYPE=Release \
  -G Ninja \
  -S /llvm-project/runtimes \
  -B /llvm-project/build-msan \
  -DCMAKE_INSTALL_PREFIX=/usr/local/instrumented/msan \
  -DLLVM_USE_SANITIZER=MemoryWithOrigins \
  -DLLVM_ENABLE_RUNTIMES='libcxx;libcxxabi’ \
  -DLIBCXXABI_USE_LLVM_UNWINDER=Off && \
  cmake --build /llvm-project/build-msan --target install && \
  rm -rf /llvm-project/

ENV MSAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer
```

MSan : CMAKE_TOOLCHAIN_FILE

```
# Link to MSAN instrumented libc++
add_compile_options($<$<COMPILE_LANGUAGE:C,CXX>:-nostdlib++>)
add_link_options($<$<COMPILE_LANGUAGE:C,CXX>:-nostdlib++>)
link_libraries("/usr/local/instrumented/msan/lib/libc++.so.1.0")
link_libraries("/usr/local/instrumented/msan/lib/libc++abi.so.1.0")
add_compile_options($<$<COMPILE_LANGUAGE:C,CXX>:-stdlib=libc++>)
add_link_options($<$<COMPILE_LANGUAGE:C,CXX>:-stdlib=libc++>)

# Compilation and linker flags to ensure proper sanitizer support
set (CMAKE_POSITION_INDEPENDENT_CODE ON)
set(MSAN_FLAGS
  -fsanitize=memory
  -fsanitize-memory-track-origins
)
add_compile_options(
  $<$<COMPILE_LANGUAGE:C,CXX>:-fno-omit-frame-pointer>
  ${MSAN_FLAGS})
add_link_options(
  $<$<COMPILE_LANGUAGE:C,CXX>:-fno-omit-frame-pointer>
  ${MSAN_FLAGS})
```

MSan : Let’s take an example

```cpp
int compute_array_sum(const std::array<int, 10> arr) {
  int sum=0;
  for (auto v : arr) { sum+=v; }
  return sum;
}
```

```dockerfile
FROM tipibuild/tipi-ubuntu-2404-clang20@sha256:...

# Preinstall dependencies
RUN mkdir -p /compute_array_sum && \
  git clone ...compute_array_sum.git /compute_array_sum && \
  cmake -DCMAKE_BUILD_TYPE=Release \
  -G Ninja \
  -S /compute_array_sum \
  -B /compute_array_sum/build \
  -DCMAKE_INSTALL_PREFIX=/usr/ && \
  cmake --build /compute_array_sum/build --target install
```

MSan : Let’s take an example

```
find_package(compute_array_sum REQUIRED)

add_executable(main test/main.cpp)
target_link_libraries(main compute_array_sum::compute_array_sum)

add_test(NAME main COMMAND $<TARGET_FILE:main>)
```

```
#include <print>
#include <array>

#include <compute_array_sum.hpp>

int main() {
  std::array<int,10> arr;
  arr[5] = 0;
  std::println(”Array sum: {} \n", compute_array_sum(arr));
  return 0;
}
```

MSan : Let’s take an example

No instrumentation

```cmake
find_package(compute_array_sum REQUIRED)

add_executable(main test/main.cpp)
target_link_libraries(main compute_array_sum::compute_array_sum)

add_test(NAME main COMMAND $<TARGET_FILE:main>)
```

```cpp
#include <print>
#include <array>

#include <compute_array_sum.hpp>

int main() {
  std::array<int,10> arr;
  arr[5] = 0;
  std::println(”Array sum: {} \n", compute_array_sum(arr));
  return 0;
}
```

MSan : Let’s get instrumentation everywhere

```
- find_package(compute_array_sum REQUIRED)

+ FetchContent_Declare(
+  compute_array_sum
+  GIT_REPOSITORY .../unittest-compute_array_sum.git
+  GIT_TAG 9cbe670)
+ FetchContent_MakeHermetic(compute_array_sum)
+ HermeticFetchContent_MakeAvailableAtBuildTime(compute_array_sum)

  add_executable(main test/main.cpp)
  target_link_libraries(main compute_array_sum::compute_array_sum)

  add_test(NAME main COMMAND $<TARGET_FILE:main>)
```

Instrumentation ✅

MSan : Let’s get instrumentation everywhere

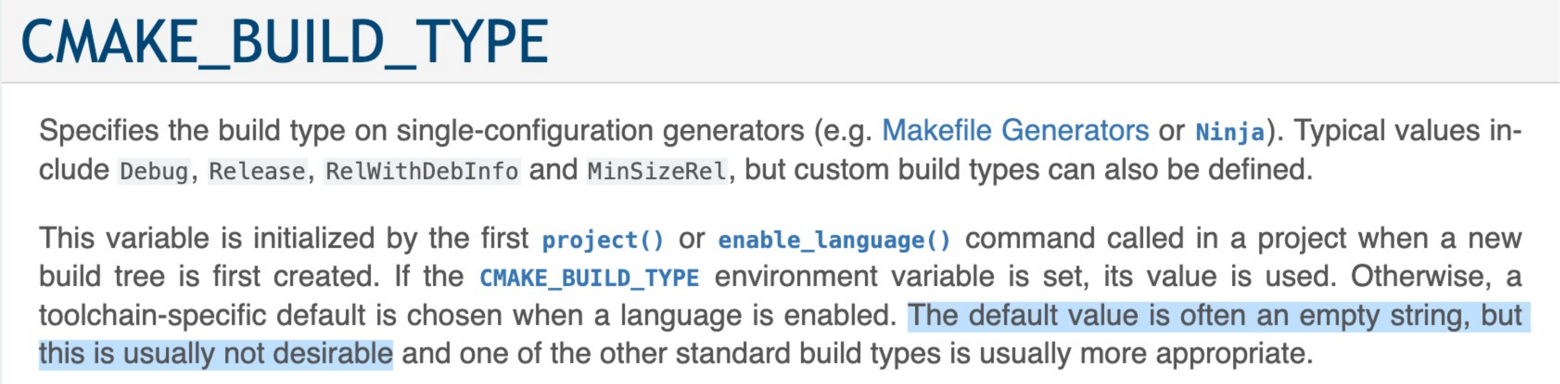
```
# HFC to the rescue
FetchContent_MakeHermetic(<name>

  HERMETIC_BUILD_SYSTEM cmake | autotools | openssl

  HERMETIC_FIND_PACKAGES <list of exposed dependencies>

  HERMETIC_TOOLCHAIN_EXTENSION <cmake code>

  HERMETIC_CMAKE_EXPORT_LIBRARY_DECLARATION <cmake code>

  SBOM_LICENSE "<SPDX-Identifier>"
  SBOM_SUPPLIER "<Author>"

)
```

⭐ **Please Star it on Github : https://github.com/tipi-build/hfc**

MSan Release vs Debug Symbols

```
==222900==WARNING: MemorySanitizer: use-of-uninitialized-value
  #0 0xf in compute_array_sum(std::__1::array<int, 10ul>) (/main+0x1082bd)compute_array_sum.cpp:6:3
  #1 0xf in main (/main+0xce5ed)test/main.cpp:12:36
  #2 0xf in __libc_start_call_main csu/../sysdeps/nptl/libc_start_call_main.h:58:16
  #3 0xf in __libc_start_main csu/../csu/libc-start.c:360:3
  #4 0xf in _start (/main+0x33494)

Uninitialized value was created by an allocation of 'agg.tmp1' in the stack frame
  #0 0xf in main (/main+0xce577)test/main.cpp:6

SUMMARY: MemorySanitizer: use-of-uninitialized-value (/main+0x1082bd)compute_array_sum.cpp:6:3 in
compute_array_sum(std::__1::array<int, 10ul>)
```

Choosing CMAKE_BUILD_TYPE



CMAKE_BUILD_TYPE

Specifies the build type on single-configuration generators (e.g. Makefile Generators or `Ninja`). Typical values include `Debug`, `Release`, `RelWithDebInfo` and `MinSizeRel`, but custom build types can also be defined.

This variable is initialized by the first `project()` or `enable_language()` command called in a project when a new build tree is first created. If the `CMAKE_BUILD_TYPE` environment variable is set, its value is used. Otherwise, a toolchain-specific default is chosen when a language is enabled. The default value is often an empty string, but this is usually not desirable and one of the other standard build types is usually more appropriate.

Choosing CMAKE_BUILD_TYPE

"": No Optimizations, No Debug Symbols

Release: Optimizations, No Debug Symbols
Debug: No Optimizations, Debug Symbols
MinSizeRel: Optimizations for Size, No Debug Symbols
RelWithDebInfo: Optimizations, Debug Symbols

→ **Rationales**

- Release & MinSizeRel have no Debug Symbols to ensure binaries only contain necessary data for operations
- Debug has no optimizations because optimizations can make debugging surprising

→ **Typical Pitfalls when considering only Debug & Release**

- Developers test most of their time in Debug
- Release crashes are undebuggable

→ **Hot Take :** RelWithDebInfo is the only sane option

Handling Debug Symbols Bloat

Problem with many very long template<> symbols

GNU LD: `relocation truncated to fit: R_X86_64_32 against `.debug_str'`
mold: `(.debug_str): mergeable section too large`

→ Can be avoided with :
Either `-g -gsplit-dwarf`
Or `-gsimple-template-names`

Debug Symbols Impacts

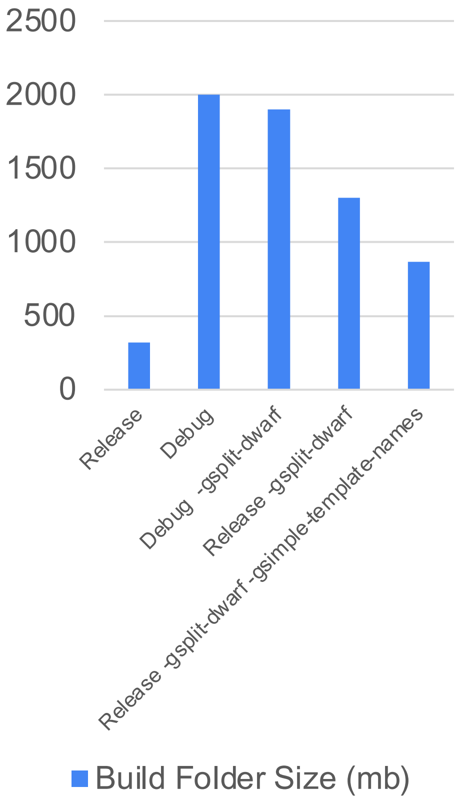
Building Boost 1.85.0 and Unit Tests with `-gsplit-dwarf,` generates a .o and a .dwo

.o

- .debug_info links to the .dwo
- Index for symbols availability in the .dwo
  .debug_gnu_pubnames
  .debug_gnu_pubtypes

.dwo

- .debug_info.dwo
- .debug_str.dwo
- .debug_rnglists.dwo
- .debug_loclists.dwo
- .debug_str_offsets.dwo
- .debug_abbrev.dwo



tipi.build by EngFlow

Precompiled : Valgrind as last resort

When **precompiled dependencies** are **unavoidable**

→ Runtime Instrumentation via Dynamic Binary Instrumentation

→ Slows down execution by 20x

→ Complex Diagnostics

```
# Enable Valgrind on CTest
set(CMAKE_TEST_LAUNCHER /usr/bin/valgrind;--track-origins=yes)
```

ASan + LSan + UBSan

AddressSanitizer + LeakSanitizer

→ Overrides malloc() and free() at runtime
Same shadowing strategy than MSAN, checked on Memory Read.

→ Slows down execution only by 2x

→ Supported on Windows (Asan only), Linux, macOS

UndefinedBehaviourSanitizer

→ Compiler Instrumentation
e.g. Adds non-null checks before any pointer dereference, check integer overflow on addition...

→ Supported on Windows, Linux, macOS

ASan + LSan + UBSan : CMAKE_TOOLCHAIN_FILE

```
# Compilation and linker flags to ensure proper sanitizer support
set (CMAKE_POSITION_INDEPENDENT_CODE ON)

set(ASAN_FLAGS
  -fsanitize=address
)

set(UBSAN_FLAGS
  -fsanitize=undefined
  -fno-sanitize-merge
)

add_compile_options(
  $<$<COMPILE_LANGUAGE:C,CXX>:-fno-omit-frame-pointer>
  ${ASAN_FLAGS}
  ${UBSAN_FLAGS})
add_link_options(
  $<$<COMPILE_LANGUAGE:C,CXX>:-fno-omit-frame-pointer>
  ${ASAN_FLAGS}
  ${UBSAN_FLAGS})
```

TSan

ThreadSanitizer

→ Code Instrumentation
For every memory access (read or write) + synchronization primitive

→ Each thread is assigned a vector clock, on communication (e.g. via a mutex) the vector clock are synchronized

→ A data race is reported when 2 accesses to the same memory location have no happens-before relation, and one of them is a write

→ Performance slow down of 5x-15x

Setting up TSAN : CMAKE_TOOLCHAIN_FILE

```
# Compilation and linker flags to ensure proper sanitizer support
set (CMAKE_POSITION_INDEPENDENT_CODE ON)

add_compile_options($<$<COMPILE_LANGUAGE:C,CXX>:-fsanitize=thread>)
add_link_options($<$<COMPILE_LANGUAGE:C,CXX>:-fsanitize=thread>)

add_compile_options(
  $<$<COMPILE_LANGUAGE:C,CXX>:-fno-omit-frame-pointer>
)
add_link_options(
  $<$<COMPILE_LANGUAGE:C,CXX>:-fno-omit-frame-pointer>
)
```

TSan : Let’s take an example

```cpp
class Writer {
public:
  void write(std::string word) {
    words_.push_back(std::move(word));
  }
private:
  std::vector<std::string> words_;
};

void threadedWork(int thread_id, Writer *writer) {
  for (auto count : std::ranges::iota_view(0, 100)) {
     writer->write(std::format("Thread {} count {}", thread_id, count));
  }
}

Writer w;
std::thread t1{[&](){threadedWork(1,&w);}};
std::thread t2{[&](){threadedWork(2,&w);}};
...
```

TSan : Detects the data race

```
WARNING: ThreadSanitizer: data race (pid=11586)
  Read of size 8 at 0x00016b07b128 by thread T2:
    #0 std::__1::vector<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char>>, std::__1::allocator<std::__1::basic_string<char, std::__1::char_traits<char>,
std::__1::allocator<char>>>>::push_back[abi:ne190102](std::__1::basic_string<char,
std::__1::char_traits<char>, std::__1::allocator<char>>&&) <null> (exe1_tsan:arm64+0x1000020b8)
    #1 Writer::write(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char>>)
<null> (exe1_tsan:arm64+0x1000018fc)
    #2 threadedWork(int, Writer*) <null> (exe1_tsan:arm64+0x1000015f4)
    #3 decltype(std::declval<void (*)(int, Writer*)>()(std::declval<int>(), std::declval<Writer*>()))
std::__1::__invoke[abi:ne190102]<void (*)(int, Writer*), int, Writer*>(void (*&&)(int, Writer*), int&&,
Writer*&&) <null> (exe1_tsan:arm64+0x100006788)
    #4 void std::__1::__thread_execute[abi:ne190102]<std::__1::unique_ptr<std::__1::__thread_struct,
std::__1::default_delete<std::__1::__thread_struct>>, void (*)(int, Writer*), int, Writer*, 2ul,
3ul>(std::__1::tuple<std::__1::unique_ptr<std::__1::__thread_struct,
std::__1::default_delete<std::__1::__thread_struct>>, void (*)(int, Writer*), int, Writer*>&,
std::__1::__tuple_indices<2ul, 3ul>) <null> (exe1_tsan:arm64+0x100006654)
    #5 void*
std::__1::__thread_proxy[abi:ne190102]<std::__1::tuple<std::__1::unique_ptr<std::__1::__thread_struct,
std::__1::default_delete<std::__1::__thread_struct>>, void (*)(int, Writer*), int, Writer*>>(void*) <null>
(exe1_tsan:arm64+0x100005868)
```

(+10 more screens)

Productivity & Safety

Really want to avoid safety bugs ?
Sanitizers requires you to add at least 3 new build types to a project existing targets

→ **New Build Types for :**

1. Msan
2. ASan + LSan + UBSan
3. TSan

+ Adding clang-tidy on top of your existing build

We optimize this, that’s one of the main reasons why we developed our remote-execution engine in CMake RE.

Clang TSA

Thead Safety Analysis

→ Annotations allowing compile-time checks for data races

→ No Runtime Performance Impact

→ Works on all platform with libc++

TSA : Improve our example

```cpp
#include "tsa.h"

class Writer {
public:
  void write(std::string word) {
    words_.push_back(std::move(word));
  }
private:
  std::mutex mutex_;
  std::vector<std::string> words_ GUARDED_BY(mutex_);
};

void threadedWork(int thread_id, Writer *writer) {
  for (auto count : std::ranges::iota_view(0, 100)) {
    writer->write(std::format("Thread {} count {}", thread_id, count));
  }
}

Writer w;
std::thread t1{[](){threadedWork(1,&w);}}; ...
```

TSA : Improve our example

```
/usr/bin/c++ -D_LIBCPP_ENABLE_THREAD_SAFETY_ANNOTATIONS -Wthread-safety -std=c++20
exe2.cpp -o exe2

exe2.cpp:13:9: warning: reading variable 'words_' requires holding mutex 'mutex_' [-
Wthread-safety-analysis]

   13 |         words_.push_back(std::move(word));

      |         ^
```

Clang TSA

Thead Safety Analysis

→Requires Clang and Compilation warning: `-Wthread-safety`
→std:: types support: `-D_LIBCPP_ENABLE_THREAD_SAFETY_ANNOTATIONS`
→boost:: types support: `-DBOOST_THREAD_ENABLE_THREAD_SAFETY_ANALYSIS`

⭐ **For Windows FetchContent libcxx :**
- https://github.com/Orphis/LibcxxFetchContentTest

➡️ Learn more about Clang TSA with Florent Castelli (@Orphis) at Stockholm Cpp Meetup
- https://www.youtube.com/watch?v=VAnlVjrDouA

Getting all the sanitizers

→ Enhance your codebase safety today : CMake RE Local Containerized builds **is free** to use.

→ Launch your sanitizers builds from macOS, Windows, Linux

```
cmake-re -S . -B build/msan \
  -DCMAKE_TOOLCHAIN_FILE=environments/linux-ubuntu-2404-clang20-msan.cmake

cmake-re --build build/msan --run-test main --monitor
```

⭐ **Get the free cmake-re sanitizers environments and example :**
https://github.com/tipi-build/example-cmake-re-sanitizers

→ For faster builds+tests with `-j1000+` cores, sign-up on tipi.build & contact us : hello@tipi.build !