Modern build systems

Build can be agile too

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

Build automation

Automating various development tasks

- Compiling source code to binaries
- Linking binaries
- Processing resources
- Running tests
- Deployment
- Documentation generation

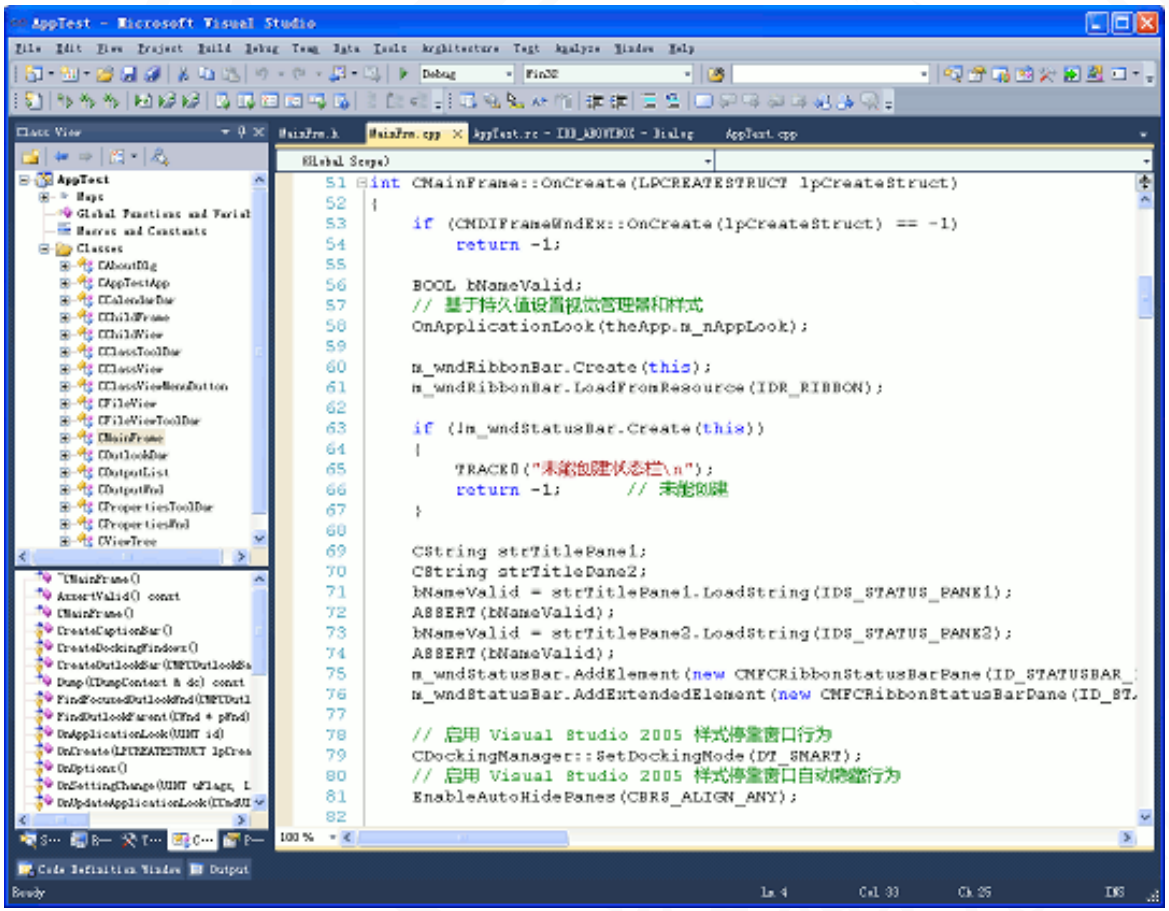
Build automation

Why automating builds ?

- Huge time saver
- Faster *edit*/*compile*/*link*/*run* cycles
- Bad builds less likely
- Reproducible builds
- Multi-platform builds
- Versioned with the source code

IDE

But *Visual Studio*/*Eclipse*/*XCode*/*<put here your favorite IDE>* build things for me !



IDE

Using IDE build not always ideal. They tend to make build

- harder to reproduce
- separated from the source code
- force to use one IDE
- hard to fully automated complex builds
- very hard to do multi-platform build

The famous "it works on *my* computer"

Table of Contents

waf

Example of modern, last-generation build system : *waf*

- Support multi-platform build
- No software to install
- Easy to learn
- Fast
- Extensible
- Integrate well with IDE
- Very good documentation
- Small (*86 Ko*)

waf

waf is a Python application

- Build described in Python
- Not one more specialized language to learn
- A complete, full-featured language

waf users

waf is young, yet popular already

- Companies like *Cisco*, *Avalanche Studios* or *MobileEyes* for *all* their projects
- Open-sources projects like *Samba*, *NodeJS*, ...

Building a simple program

Let's build a very simple project

1. a directory with one *C* file
2. we copy *waf* in the directory
3. we create a *wscript* file in the directory

Building a simple program

The file hierarchy

```
hello
 |-- Main.c
 |-- waf
 |-- wscript
```

Build description

The *wscript* file

```
#! /usr/bin/env python

VERSION = '1.0.0'
APPNAME = 'hello'

top, out = '.', 'build'

def options(context):
  context.load('compiler_cc')

def configure(context):
  context.load('compiler_cc')

def build(context):
  context.program(
    name     = 'hello',
    target   = 'hello',
    source   = 'Main.c'
  )
```

- *options* allows you to define options
- *configure* defines the configuration step
- *build* defines the build step

Starting a build

First, you need to call *configure* command.

```
waf configure
```

It will return this

```
Setting top to                           : /home/alex/01-hello
Setting out to                           : /home/alex/01-hello/build
Checking for 'gcc' (c compiler)          : ok
'configure' finished successfully (0.045s)
```

Starting a build

The *configure* step checks

- if all the required tools are installed
- if all the required libraries are installed

You do this only once

Running a build

Then, you need to call command *build*

```
waf build
```

It will return this

```
Waf: Entering directory '/home/alex/01-hello/build'
[1/2] c: Main.c -> build/Main.c.0.o
[2/2] cprogram: build/Main.c.0.o -> build/hello
Waf: Leaving directory '/home/alex/01-hello/build'
'build' finished successfully (0.058s)
```

Running a build

This step have done this

1. Compiled the source file
2. Created a binary
3. All this done in the *build* directory

It would work on Windows, MacOS, . . . with most *C* compilers

Cleaning a build

If you call the command *clean*

```
waf clean
```

It will return this

```
'clean' finished successfully (0.007s)
```

And remove all the work done by *build* but not by *configure*

Cleaning a build

If you call the command *distclean*

```
waf distclean
```

It will return this

```
'distclean' finished successfully (0.002s)
```

it will remove all the work done by *build* & *configure*

More source files

What if I have several source files ?

- Many headers files in *include* directory
- Many source files in *src* directory

Separate compilation

The file hierarchy

```
super-hello
  |
  |-- include
  |      |-- Error.h
  |      |-- HelloApplication.h
  |
  |-- src
  |      |-- Error.c
  |      |-- HelloApplication.c
  |      |-- Main.c
  |
  |-- waf
  |-- wscript
```

Build description

The *wscript* file

```
#! /usr/bin/env python

VERSION = '1.0.0'
APPNAME = 'super-hello'

top, out = '.', 'build'

def options(context):
  context.load('compiler_cc')

def configure(context):
  context.load('compiler_cc')

def build(context):
  context.program(
    name     = 'hello',
    target   = 'hello',
    source   = context.path.ant_glob('src/*.c'),
    includes = 'include'
  )
```

Running a build

The build will show this

```
Waf: Entering directory '/home/alex/02-super-hello/build'
[1/4] c: src/Error.c -> build/src/Error.c.0.o
[2/4] c: src/HelloApplication.c -> build/src/HelloApplication.c.0.o
[3/4] c: src/Main.c -> build/src/Main.c.0.o
[4/4] cprogram: build/src/Error.c.0.o build/src/HelloApplication.c.0.o build/src/Mai
Waf: Leaving directory '/home/alex/02-super-hello/build'
'build' finished successfully (0.096s)
```

Dependency resolution

waf found the correct order to compile files !

- No need to manually update a list of files to build
- No need to code dependencies handling
- No need to have build files in each directories

Less maintenance work for you

A more complex project

Let's have a more complex project with

- 1 static library to read files, *read*
- 1 static library to handle dynamic array, *array*
- 2 programs using the 2 libraries, *mean*, *median*

Multiple targets

The file hierarchy

```
crapculator
   |
   |-- libs
   |      |
   |      |-- array
   |      |      |
   |      |      |-- include
   |      |      |      |-- Array.h
   |      |      |
   |      |      |-- src
   |      |             |-- Array.c
   |      |-- read
   |             |
   |             |-- include
   |             |      |-- Read.h
   |             |
   |             |-- src
   |-- tools            |-- Read.c
   |      |
   |      |-- mean
   |      |      |-- Main.c
   |      |
   |      |-- median
   |             |-- Main.c
   |-- waf
   |-- wscript
```

Multiple targets

The *wscript* file, the *build* function ⇒ 2 static libraries

```
def build(context):
  context.stlib(
    name             = 'read',
    target           = 'read',
    source           = context.path.ant_glob('libs/read/src/*.c'),
    includes         = 'libs/read/include',
    export_includes = 'libs/read/include'
  )

  context.stlib(
    name             = 'array',
    target           = 'array',
    source           = context.path.ant_glob('libs/array/src/*.c'),
    includes         = 'libs/array/include',
    export_includes = 'libs/array/include'
  )

  ...
```

Multiple targets

The *wscript* file, the *build* function $\Rightarrow$ 2 programs

```
def build(context):
    ...

    context.program(
        name     = 'mean',
        target   = 'mean',
        source   = context.path.ant_glob('tools/mean/*.c'),
        includes = 'include',
        use      = ['read', 'array']
    )

    context.program(
        name     = 'median',
        target   = 'median',
        source   = context.path.ant_glob('tools/median/*.c'),
        includes = 'include',
        use      = ['read', 'array']
    )
```

Multiple targets

The build will show this

```
Waf: Entering directory '/home/alex/examples/03-crapculator/build'
[1/8] c: libs/read/src/Read.c -> build/libs/read/src/Read.c.0.o
[2/8] c: libs/array/src/Array.c -> build/libs/array/src/Array.c.1.o
[3/8] c: tools/mean/Main.c -> build/tools/mean/Main.c.2.o
[4/8] c: tools/median/Main.c -> build/tools/median/Main.c.3.o
[5/8] cstlib: build/libs/read/src/Read.c.0.o -> build/libread.a
[6/8] cstlib: build/libs/array/src/Array.c.1.o -> build/libarray.a
[7/8] cprogram: build/tools/mean/Main.c.2.o -> build/mean
[8/8] cprogram: build/tools/median/Main.c.3.o -> build/median
Waf: Leaving directory '/home/alex/03-crapculator/build'
'build' finished successfully (0.149s)
```

Dependency resolution

waf found the correct order to compile & link !

- No need to worry about build order *at all*

Less maintenance work for you

Scripting

wscript files are Python code. Let's use Python to make the previous script shorter

```
def build(context):
  stlibList = ['read', 'array']
  for item in stlibList:
    context.stlib(
      name            = item,
      target          = item,
      source          = context.path.ant_glob('libs/%s/src/*.c' % item),
      includes        = 'libs/%s/include' % item,
      export_includes = 'libs/%s/include' % item
    )

  progList = ['mean', 'median']
  for item in progList:
    context.program(
      name     = item,
      target   = item,
      source   = context.path.ant_glob('tools/%s/*.c' % item),
      includes = 'include',
      use      = stlibList
    )
```

Scripting

By using Python as the build description language

- Not one more limited language (unlike *make*, *autoconf*, *cmake*, *ant*, *jam*, ...)
- A clean, complete language
- Full access to a large standard library
- Reading external files, accessing remote data, ...

Powerful builds are easy to write, easy to maintain

Build variants

It's quite common to need *variants* of a build

- A *debug* build, with special compiler flags and features
- A *release* build, with special compiler flags and no debug features
- Different builds for different customers
- ...

Build variants

Let's define a *debug* and a *release* variant for our project

Build variants

The *configure* function describe the 2 variants

```
def configure(context):
  context.setenv('debug')
  context.load('compiler_cc')
  context.env.CCFLAGS = ['-g']

  context.setenv('release', env=context.env.derive())
  context.env.CCFLAGS = ['-O2']
```

Build variants

An *init* function defines *build–debug* and *build–release* commands

```
def init(context):
  from waflib.Build import BuildContext, CleanContext, InstallContext, UninstallConte

  for x in 'debug_release'.split():
    for y in (BuildContext, CleanContext, InstallContext, UninstallContext):
      name = y.__name__.replace('Context','').lower()
      class tmp(y):
        cmd = name + '_' + x
        variant = x

  def buildall(context):
    import waflib.Options
    for x in ['build_debug', 'build_release']:
      waflib.Options.commands.insert(0, x)
```

Build variants

The *configure* command will show the same usual things

```
Setting top to                           : /home/alex/03-crapculator
Setting out to                           : /home/alex/03-crapculator/build
Checking for 'gcc' (c compiler)          : ok
'configure' finished successfully (0.047s)
```

Build variants

The *build* command will not work any more

```
Waf: Entering directory '/home/alex/03-crapculator/build'
Call "waf build debug" or "waf build release",
and read the comments in the wscript file!
```

Build variants

But we got a new *build–debug* command

```
Waf: Entering directory '/home/alex/03-crapculator/build/debug'
[1/8] c: libs/read/src/Read.c -> build/debug/libs/read/src/Read.c.0.o
[2/8] c: libs/array/src/Array.c -> build/debug/libs/array/src/Array.c.1.o
[3/8] c: tools/mean/Main.c -> build/debug/tools/mean/Main.c.2.o
[4/8] c: tools/median/Main.c -> build/debug/tools/median/Main.c.3.o
[5/8] cstlib: build/debug/libs/read/src/Read.c.0.o -> build/debug/libread.a
[6/8] cstlib: build/debug/libs/array/src/Array.c.1.o -> build/debug/libarray.a
[7/8] cprogram: build/debug/tools/mean/Main.c.2.o -> build/debug/mean
[8/8] cprogram: build/debug/tools/median/Main.c.3.o -> build/debug/median
Waf: Leaving directory '/home/alex/03-crapculator/build/debug'
'build_debug' finished successfully (0.211s)
```

Build variants

And a new *build–release* command

```
Waf: Entering directory '/home/alex/03-crapculator/build/release'
[1/8] c: libs/read/src/Read.c -> build/release/libs/read/src/Read.c.0.o
[2/8] c: libs/array/src/Array.c -> build/release/libs/array/src/Array.c.1.o
[3/8] c: tools/mean/Main.c -> build/release/tools/mean/Main.c.2.o
[4/8] c: tools/median/Main.c -> build/release/tools/median/Main.c.3.o
[5/8] cstlib: build/release/libs/read/src/Read.c.0.o -> build/release/libread.a
[6/8] cstlib: build/release/libs/array/src/Array.c.1.o -> build/release/libarray.a
[7/8] cprogram: build/release/tools/mean/Main.c.2.o -> build/release/mean
[8/8] cprogram: build/release/tools/median/Main.c.3.o -> build/release/median
Waf: Leaving directory '/home/alex/Boulot/03-crapculator/build/release'
'build_release' finished successfully (0.178s)
```

Mixing languages

You can decide to use different languages.

- Company's old code base in *C*
- Company's new code base in *C++*, *Java*, ...
- Using libraries written in other languages
- Language imposed by the platform (*iPhone*,*iPad* development)
- ...

Mixing languages

Let's consider the previous project, but

- Keeping the libraries in *C*
- Adding libraries in *C++*, using the *C* libraries
- The programs are in *C++*, using the *C++* libraries

Mixing languages

The file hierarchy

```
crapculator
  |
  |-- libs
  |     |
  |     |-- array
  |     |     |
  |     |     |-- include
  |     |     |     |-- Array.h
  |     |     |
  |     |     |-- src
  |     |           |-- Array.c
  |     |-- read
  |     |     |
  |     |     |-- include
  |     |     |     |-- Read.h
  |     |     |
  |     |     |-- src
  |     |           |-- Read.c
  ...
```

Mixing languages

The file hierarchy

```
crapculator
   ...
   |      |-- array++
   |      |      |
   |      |      |-- include
   |      |      |      |-- Array.hpp
   |      |      |
   |      |      |-- src
   |      |             |-- Array.cpp
   |      |-- read++
   |             |
   |             |-- include
   |             |      |-- Reader.hpp
   |             |
   |             |-- src
   |-- tools     |-- Reader.cpp
   |      |
   |      |-- mean
   |      |      |-- Main.cpp
   |      |
   |      |-- median
   |             |-- Main.cpp
   |-- waf
   |-- wscript
```

Mixing languages

The *wscript* does not change much, only declare that we intend to use *C++*

```
def options(context):
  context.load('compiler_cc')
  context.load('compiler_cxx')

def configure(context):
  context.load('compiler_cc')
  context.load('compiler_cxx')
  context.env.CCFLAGS = ['-O2']
  context.env.CXXFLAGS = ['-O2']
```

Mixing languages

The *wscript* does not change much, only declare that we intend to use *C++*

```
def build(context):
  stlibList = ['read', 'array']
  for item in stlibList:
    context.stlib(
      name            = item,
      target          = item,
      source          = context.path.ant_glob('libs/%s/src/*.c' % item),
      includes        = 'libs/%s/include' % item,
      export_includes = 'libs/%s/include' % item
    )

    context.stlib(
      name            = '%s++' % item,
      target          = '%s++' % item,
      source          = context.path.ant_glob('libs/%s++/src/*.cpp' % item),
      includes        = 'libs/%s++/include' % item,
      export_includes = 'libs/%s++/include' % item,
      use             = [ item ]
    )

  ...
```

Mixing languages

The *wscript* does not change much, only declare that we intend to use *C++*

```
def build(context):
  ...

  progList = ['mean', 'median']
  for item in progList:
    context.program(
      name     = item,
      target   = item,
      source   = context.path.ant_glob('tools/%s/*.cpp' % item),
      includes = 'include',
      use      = ['read++', 'array++']
    )
```

Mixing languages

The *configure* command will show this

```
Setting top to                           : /home/alex/04-crapculator-2
Setting out to                           : /home/alex/04-crapculator-2/build
Checking for 'gcc' (c compiler)          : ok
Checking for 'g++' (c++ compiler)        : ok
'configure' finished successfully (0.086s)
```

Mixing languages

The *build* command will show this

```
Waf: Entering directory '/home/alex/04-crapculator-2/build'
[ 1/12] c: libs/read/src/Read.c -> build/libs/read/src/Read.c.0.o
[ 2/12] cxx: libs/read++/src/Reader.cpp -> build/libs/read++/src/Reader.cpp.1.o
[ 3/12] c: libs/array/src/Array.c -> build/libs/array/src/Array.c.2.o
[ 4/12] cxx: libs/array++/src/Array.cpp -> build/libs/array++/src/Array.cpp.3.o
[ 5/12] cxx: tools/mean/Main.cpp -> build/tools/mean/Main.cpp.4.o
[ 6/12] cxx: tools/median/Main.cpp -> build/tools/median/Main.cpp.5.o
[ 7/12] cstlib: build/libs/read/src/Read.c.0.o -> build/libread.a
[ 8/12] cstlib: build/libs/array/src/Array.c.2.o -> build/libarray.a
[ 9/12] cxxstlib: build/libs/read++/src/Reader.cpp.1.o -> build/libread++.a
[10/12] cxxstlib: build/libs/array++/src/Array.cpp.3.o -> build/libarray++.a
[11/12] cxxprogram: build/tools/mean/Main.cpp.4.o -> build/mean
[12/12] cxxprogram: build/tools/median/Main.cpp.5.o -> build/median
Waf: Leaving directory '/home/alex/04-crapculator-2/build'
'build' finished successfully (0.307s)
```

Mixing languages

Once again, it's all automatic ! *waf* support many languages

Code generation

Programs creating code !

- Lexical analysers
- Parsers
- Large, complex table
- Large finite state automaton
- ...

The idea is to avoid writing by hand large, boring and error-prone code, like a big table

Pearson hashing

A basic example : *Pearson hashing*

```
def pearsonHash(message):
  hashValue = 0
  for char in message:
    hashValue = T[hashValue ^ char]

  return hashValue
```

Very cheap yet effective hashing scheme

Pearson hashing

The table T

- 256 values from $[0, 255]$
- each value only one time
- it's called a *permutation*

C implementation

Let's implement this in *C*

- Program *permutgen* generate permutation tables in a *C* file
- We will generate a *permutTable.c* file with *permutgen*
- Program *hash* use the permutation table defined in *permutTable.c*

C implementation

The file hierarchy

```
hash
 |
 |-- tools
 |     |
 |     |-- hash
 |     |     |-- Main.c
 |     |
 |     |-- permutgen
 |           |-- Main.c
 |-- waf
 |-- wscript
```

C implementation

The *wscript* file, *build* function

```
def build(context):
  context.program(
    name     = 'permutgen',
    target   = 'permutgen',
    source   = context.path.ant_glob('tools/permutgen/*.c'),
  )

  context.add_group()

  def genPermutTable(task):
    tool   = task.inputs[0].abspath()
    target = task.outputs[0].abspath()
    cmd    = '%s %s permutTable' % (tool, target)
    return task.exec_command(cmd)

  context(
    rule   = genPermutTable,
    source = 'permutgen',
    target = 'permutTable.c'
  )

  context.program(
    name     = 'hash',
    target   = 'hash',
    source   = 'permutTable.c tools/hash/Main.c',
  )
```

C implementation

The *build* command will show this

```
Waf: Entering directory '/home/alex/05-hash/build'
[1/6] c: tools/permutgen/Main.c -> build/tools/permutgen/Main.c.0.o
[2/6] cprogram: build/tools/permutgen/Main.c.0.o -> build/permutgen
[3/6] permutTable.c: build/permutgen -> build/permutTable.c
[4/6] c: tools/hash/Main.c -> build/tools/hash/Main.c.2.o
[5/6] c: build/permutTable.c -> build/permutTable.c.2.o
[6/6] cprogram: build/permutTable.c.2.o build/tools/hash/Main.c.2.o -> build/hash
Waf: Leaving directory '/home/alex/05-hash/build'
'build' finished successfully (0.776s)
```

C implementation

The *C* code generated by *permutgen*

```
const int permutTable[256] = {
  0x12, 0x74, 0x41, 0x07, 0xdc, 0xbf, 0x7a, 0x0e,
  0x6b, 0x69, 0xf1, 0x56, 0x25, 0x48, 0x70, 0x7b,
  0xe1, 0xa4, 0x2f, 0x26, 0x06, 0xe8, 0x6f, 0x14,
  0xcb, 0xc3, 0xf2, 0x7f, 0x7e, 0xb9, 0xc6, 0x36,
  0x33, 0xe4, 0x60, 0x2e, 0x18, 0xc5, 0xde, 0x86,
  0xa0, 0xed, 0xfa, 0xc1, 0x49, 0x05, 0x65, 0xaf,
  0x2a, 0x42, 0xe0, 0x96, 0x0b, 0xe5, 0xb2, 0xe3,
  0x85, 0xce, 0xb5, 0x5f, 0xd0, 0x20, 0x5e, 0x5d,
  0x95, 0xe7, 0x87, 0x54, 0xef, 0xf7, 0x75, 0x53,
  0x68, 0x11, 0x9e, 0x3d, 0x40, 0x99, 0x59, 0xa5,
  0x6d, 0x43, 0xa8, 0xab, 0x10, 0xd3, 0x51, 0x15,
  0x1f, 0x55, 0xe6, 0xb8, 0xba, 0x83, 0x3c, 0xbd,
  0x84, 0xa1, 0x3e, 0x24, 0x21, 0x0c, 0xd8, 0x13,
  0x4d, 0x72, 0x77, 0x03, 0xb7, 0x8e, 0xca, 0x27,
  0xf6, 0x80, 0x19, 0x6a, 0xe2, 0xd4, 0x9b, 0xa3,
  0x0d, 0x7c, 0x1d, 0x34, 0x00, 0x1e, 0x93, 0x2c,
  0xc0, 0x6e, 0xbc, 0xd5, 0x32, 0x50, 0x38, 0xf5,
  0x31, 0x1c, 0x90, 0x61, 0xd6, 0x16, 0x8f, 0x62,
  0x02, 0x8d, 0xdf, 0x57, 0x9c, 0x52, 0x35, 0x7d,
  0xd7, 0x46, 0x47, 0xeb, 0x01, 0x30, 0x78, 0xfd,
  0xbe, 0xbb, 0x3a, 0xa9, 0x39, 0x23, 0xc2, 0xc9,
  0x4e, 0xaa, 0x17, 0xb3, 0x8c, 0xec, 0x73, 0x44,
  ...
```