

Boostache Exposed

the internals of Boost's template engine



ciere consulting

Michael Caisse

michael.caisse@ciere.com | follow @MichaelCaisse
Copyright © 2015





boostache



boostache



boostache

boo { stache

boo { stache }



caterpillar

Part I

Introduction

C++Now 2014 Library in a Week Challenge

What is it?

C++ Template Engine

What is it?

C++ Template Engine

What!!

What is it?

Boostache is a text template processing engine.

Takes in a document (*the template*) and a data source and generates an output document based on values from the data source

What is it?

Boostache is a text template processing engine.

Takes in a document (*the template*) and a data source and generates an output document based on values from the data source

Example Libraries

- ▶ Mustache
- ▶ Handlebars
- ▶ ctemplate
- ▶ Dust
- ▶ Django
- ▶ Jade
- ▶ Jinja

Original Cast

- ▶ Michal Bukovsky
- ▶ Michael Caisse
- ▶ Jeff Garland
- ▶ Jeroen Habraken
- ▶ Kevin Harris
- ▶ Dan Nuffer

Design Goals

Design Goals (morning session):

- ▶ Integrate STL well
- ▶ Type Safety
- ▶ Template parsing error handling
- ▶ Provide customization of markup language
- ▶ Make configuration simple
- ▶ Any data structure convertible to text should work

Features

- ▶ “Compiled templates” (for speed)

Design Decisions

- ▶ Use C++11

Design Goals

Design Goals (morning session):

- ▶ Integrate STL well
- ▶ Type Safety
- ▶ Template parsing error handling
- ▶ Provide customization of markup language
- ▶ Make configuration simple
- ▶ Any data structure convertible to text should work

Features

- ▶ “Compiled templates” (for speed)

Design Decisions

- ▶ Use C++11

Design Goals

Design Goals (morning session):

- ▶ Integrate STL well
- ▶ Type Safety
- ▶ Template parsing error handling
- ▶ Provide customization of markup language
- ▶ Make configuration simple
- ▶ Any data structure convertible to text should work

Features

- ▶ “Compiled templates” (for speed)

Design Decisions

- ▶ Use C++11

Data Model Design Discussion

- ▶ Force a model on the user.
 - ▶ hash of strings to strings
 - ▶ property tree
 - ▶ JSON tree
- ▶ Accept a user defined model

Data Model Design Discussion

- ▶ Force a model on the user.
 - ▶ hash of strings to strings
 - ▶ property tree
 - ▶ JSON tree
- ▶ Accept a user defined model

Data Model Design Discussion

Accept **any** user defined model

Following examples are for the Mustache Man pages

Mustache Example

Template

```
Hello {{name}}  
You have just won {{value}} dollars!  
{{#in_ca}}  
Well, {{taxed_value}} dollars, after taxes.  
{{/in_ca}}
```

Data

```
{  
  "name": "Chris",  
  "value": 10000,  
  "taxed_value": 10000 - (10000 * 0.4),  
  "in_ca": true  
}
```

Result

```
Hello Chris  
You have just won 10000 dollars!  
Well, 6000.0 dollars, after taxes.
```

Mustache Example

Template

```
{{#repo}}  
  <b>{{name}}</b>  
{{/repo}}
```

Data

```
{  
  "repo": [  
    { "name": "resque" },  
    { "name": "hub" },  
    { "name": "rip" }  
  ]  
}
```

Result

```
<b>resque</b>  
<b>hub</b>  
<b>rip</b>
```


Using Boostache - Example 1

```
using map_t = std::map<std::string, std::string>;

int main()
{
    // template
    std::string input("My name is {{name}}. I am {{age}} years old.");

    // data
    map_t data = { { "name" , "Jeroen" },
                   { "age"  , "42"   }
    };

    using boostache::load_template;
    using boostache::format::stache;

    auto iter = input.begin();
    auto templ = load_template<stache>(iter, input.end());

    std::stringstream stream;
    boostache::generate(stream, templ, data);

    std::cout << stream.str();
}
```

Using Boostache - Example 1

```
using map_t = std::map<std::string, std::string>;

int main()
{
    // template
    std::string input("My name is {{name}}. I am {{age}} years old.");

    // data
    map_t data = { { "name" , "Jeroen" },
                   { "age"  , "42"   }
    };

    using boostache::load_template;
    using boostache::format::stache;

    auto iter = input.begin();
    auto templ = load_template<stache>(iter, input.end());

    std::stringstream stream;
    boostache::generate(stream, templ, data);

    std::cout << stream.str();
}
```

Using Boostache - Example 1

```
using map_t = std::map<std::string, std::string>;

int main()
{
    // template
    std::string input("My name is {{name}}. I am {{age}} years old.");

    // data
    map_t data = { { "name" , "Jeroen" },
                   { "age"  , "42"   }
    };

    using boostache::load_template;
    using boostache::format::stache;

    auto iter = input.begin();
    auto templ = load_template<stache>(iter, input.end());

    std::stringstream stream;
    boostache::generate(stream, templ, data);

    std::cout << stream.str();
}
```

Using Boostache - Example 1

```
using map_t = std::map<std::string, std::string>;

int main()
{
    // template
    std::string input("My name is {{name}}. I am {{age}} years old.");

    // data
    map_t data = { { "name" , "Jeroen" },
                   { "age"  , "42"   }
    };

    using boostache::load_template;
    using boostache::format::stache;

    auto iter = input.begin();
    auto templ = load_template<stache>(iter, input.end());

    std::stringstream stream;
    boostache::generate(stream, templ, data);

    std::cout << stream.str();
}
```

Using Boostache - Example 1

```
using map_t = std::map<std::string, std::string>;

int main()
{
    // template
    std::string input("My name is {{name}}. I am {{age}} years old.");

    // data
    map_t data = { { "name" , "Jeroen" },
                   { "age"  , "42"    }
    };

    using boostache::load_template;
    using boostache::format::stache;

    auto iter = input.begin();
    auto templ = load_template<stache>(iter, input.end());

    std::stringstream stream;
    boostache::generate(stream, templ, data);

    std::cout << stream.str();
}
```

Using Boostache - Example 2

```
using item_t = std::map<std::string, std::string>;
using item_list_t = std::vector<item_t>;
using invoice_t = std::map<std::string, item_list_t>;

int main()
{
    std::string input( "Invoice"
        "\n"
        "{{#lines}}"
        "  {{item_code}}  {{description}}  {{amount}}\n"
        "{{{/lines}}"
    );

    item_list_t invoice_items = {
        { {"item_code"      , "1234"},
          {"description"    , "Jolt"},
          {"amount"         , "$23"} },
        { {"item_code"      , "1235"},
          {"description"    , "computer"},
          {"amount"         , "$9"}  }
    };

    invoice_t invoice = {{"lines" , invoice_items}};
```

Using Boostache - Example 2

```
using item_t = std::map<std::string, std::string>;
using item_list_t = std::vector<item_t>;
using invoice_t = std::map<std::string, item_list_t>;

int main()
{
    std::string input( "Invoice"
                       "\n"
                       "{{#lines}}"
                       "  {{item_code}}  {{description}}  {{amount}}\n"
                       "{{{/lines}}"
                       );

    item_list_t invoice_items = {
        { {"item_code"      , "1234"},
          {"description"    , "Jolt"},
          {"amount"         , "$23"} },
        { {"item_code"      , "1235"},
          {"description"    , "computer"},
          {"amount"         , "$9"}  }
    };

    invoice_t invoice = {"lines" , invoice_items};
```

Using Boostache - Example 2

```
using item_t = std::map<std::string, std::string>;
using item_list_t = std::vector<item_t>;
using invoice_t = std::map<std::string, item_list_t>;

int main()
{
    std::string input( "Invoice"
                       "\n"
                       "{{#lines}}"
                       "  {{item_code}}  {{description}}  {{amount}}\n"
                       "{{{/lines}}"
                       );

    item_list_t invoice_items = {
        { {"item_code"      , "1234"},
          {"description"    , "Jolt"},
          {"amount"         , "$23"} },
        { {"item_code"      , "1235"},
          {"description"    , "computer"},
          {"amount"         , "$9"}  }
    };

    invoice_t invoice = {{"lines" , invoice_items}};
```


Using Boostache - Example 2

```
using boostache::load_template;
using boostache::format::stache;

auto iter = input.begin();
auto templ = load_template<stache>(iter, input.end());

std::stringstream stream;
boostache::generate(stream, templ, invoice);

std::cout << stream.str();
}
```

Using Boostache - Example 2

```
using boostache::load_template;
using boostache::format::stache;

auto iter = input.begin();
auto templ = load_template<stache>(iter, input.end());

std::stringstream stream;
boostache::generate(stream, templ, invoice);

std::cout << stream.str();
}
```

Using Boostache - Example 3

```
std::string input(
    "Invoice {{invoice_number}}"
    "\n"
    "{{# company}}"
    "Company: {{name}}\n"
    "        {{street}}\n"
    "        {{city}}, {{state}}  {{zip}}\n"
    "{{/ company}}"
    "-----\n"
    "{{#lines}}"
    "  {{item_code}}  {{description}}  {{amount}}\n"
    "{{/lines}}"
);
```

Using Boostache - Example 3

```
using boost::spirit::extended_variant;

struct value_t;
using object_t = std::map<std::string, value_t>;
using list_t = std::vector<value_t>;

struct value_t : extended_variant< std::string
                                   , object_t
                                   , list_t
                                   >
{
    value_t() : base_type() {}
    value_t(std::string const & rhs) : base_type(rhs) {}
    value_t(char const * rhs) : base_type(std::string{rhs}) {}
    value_t(object_t const & rhs) : base_type(rhs) {}
    value_t(list_t const & rhs) : base_type(rhs) {}
};
```


Using Boostache - Example 3

```
using boostache::load_template;
using boostache::format::stache;

auto iter = input.begin();
auto templ = load_template<stache>(iter, input.end());

std::stringstream stream;
boostache::generate(stream, templ, invoice);
```

Using Boostache - Example 4

```
std::string input(  
    "Hello {{first_name}} {{last_name}} - \n\n"  
    "Congratulations on the acceptance of {{library}} "  
    "to Boost!"  
    "\n"  
);  
  
author sue = { "Jones", "Sue", "sue@jones.net", "Wicket" };  
  
using boostache::load_template;  
using boostache::format::stache;  
  
auto iter = input.begin();  
auto templ = load_template<stache>(iter, input.end());  
  
std::stringstream stream;  
boostache::generate(stream, templ, sue);
```

Using Boostache - Example 4

```
struct author
{
    std::string last_name;
    std::string first_name;
    std::string email;
    std::string library;
};

std::string input(
    "Hello {{first_name}} {{last_name}} - \n\n"
    "Congratulations on the acceptance of {{library}} "
    "to Boost!"
    "\n"
);

author sue = { "Jones", "Sue", "sue@jones.net", "Wicket" };
```


Using Boostache - Example 4

```
struct author
{
    std::string last_name;
    std::string first_name;
    std::string email;
    std::string library;
};

BOOST_FUSION_ADAPT_STRUCT(
    author,
    (std::string , last_name)
    (std::string , first_name)
    (std::string , library)
)
```

Using Boostache - Example 5

```
std::string input(
    "Invoice {{invoice_number}}"
    "\n"
    "{{# company}}"
    "Company: {{name}}\n"
    "    {{street}}\n"
    "    {{city}}, {{state}}  {{zip}}\n"
    "{{/ company}}"
    "-----\n"
    "{{#lines}}"
    "  {{item_code}}  {{description}}  {{amount}}\n"
    "{{/lines}}"
);
```

Using Boostache - Example 5

```
struct company
{
    std::string name;
    std::string street;
    std::string city;
    std::string state;
    std::string zip;
};

struct line
{
    std::string item_code;
    std::string description;
    std::string amount;
};

struct invoice
{
    int invoice_number;
    company company_;
    std::vector<line> lines;
};
```

Using Boostache - Example 5

```
BOOST_FUSION_ADAPT_STRUCT(  
    company,  
    (std::string , name)  
    (std::string , street)  
    (std::string , city)  
    (std::string , state)  
    (std::string , zip)  
)  
  
BOOST_FUSION_ADAPT_STRUCT(  
    line,  
    (std::string , item_code)  
    (std::string , description)  
    (std::string , amount)  
)  
  
BOOST_FUSION_ADAPT_STRUCT(  
    invoice,  
    (int , invoice_number)  
    (company , company_)  
    (std::vector<line> , lines)  
)
```

Using Boostache - Example 5

```
using boostache::load_template;  
using boostache::format::stache;  
  
auto iter = input.begin();  
auto templ = load_template<stache>(iter, input.end());  
  
std::stringstream stream;  
boostache::generate(stream, templ, invoice_);
```

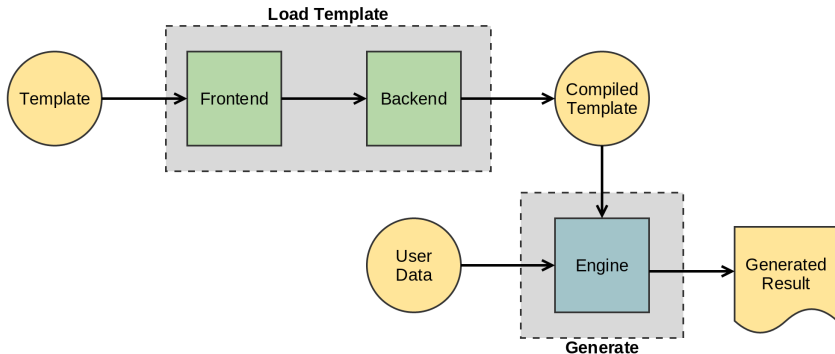
Using Boostache - Example 6

```
using boostache::load_template;  
using boostache::format::django;  
  
auto iter = input.begin();  
auto templ = load_template<django>(iter, input.end());  
  
std::stringstream stream;  
boostache::generate(stream, templ, data);
```

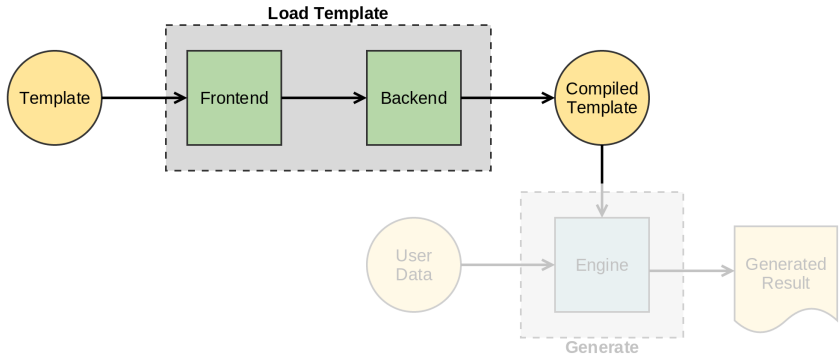
Part II

Diving In

Starting at the Top



The Frontend



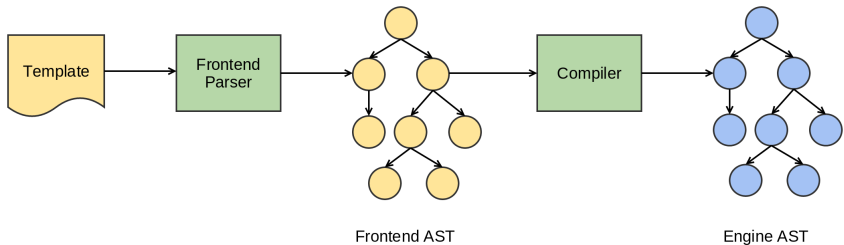
The Frontend

```
using boostache::load_template;
using boostache::format::stache;

auto iter = input.begin();
auto templ = load_template<stache>(iter, input.end());

std::stringstream stream;
boostache::generate(stream, templ, sue);
```

The Frontend



Starting Out - load_template

```
template <typename Format, typename Iterator>
inline vm::ast::node load_template( Iterator & begin
                                   , Iterator const & end)
{
    return backend::compile(frontend::parse<Format>(begin,end));
}

template <typename Format>
inline vm::ast::node load_template(std::istream & input)
{
    return backend::compile(frontend::parse<Format>(input));
}
```

Frontend Selection

```
namespace boost { namespace boostache { namespace format
{
    struct stache
    {
        template <typename Iterator>
        using grammar_t = frontend::stache::grammar<Iterator>;

        using ast_t = frontend::stache::ast::root;
        using skipper_t = boost::spirit::qi::space_type;
    };
}}}
```

Which Parser?

“Could use boost::spirit for parsing or maybe regex is enough”

Day 1 discussion notes LiaW 2014

Frontend Parse

```
template <typename Format, typename Iterator>
typename Format::ast_t parse(Iterator & begin, Iterator const & end)
{
    typename Format::ast_t ast;
    typename Format::template grammar_t<Iterator> grammar;

    if(!boost::spirit::qi::phrase_parse( begin, end
                                           , grammar
                                           , typename Format::skipper_t{}
                                           , ast ))
    {
        ast = typename Format::ast_t{};
    }
    return ast;
}
```

Organization:

- ▶ frontend
 - ▶ stache
 - ▶ ast.hpp
 - ▶ ast_adapted.hpp
 - ▶ grammar.hpp
 - ▶ grammar_def.hpp
 - ▶ printer.hpp

*users need
fusion
declaration
definition*

Organization:

- ▶ frontend
 - ▶ stache
 - ▶ ast.hpp
 - ▶ ast_adapted.hpp
 - ▶ grammar.hpp
 - ▶ grammar_def.hpp
 - ▶ printer.hpp
- users need
fusion
declaration
definition*

stache Parser

```
qi::rule<Iterator, ast::node_list(), qi::space_type>
    node_list
    ;

qi::rule<Iterator, ast::node(), qi::space_type>
    stache_node
    ;

qi::rule<Iterator, ast::identifier(), qi::space_type>
    identifier
    ;

qi::rule<Iterator, ast::comment(), qi::space_type>
    comment
    ;

qi::rule<Iterator, ast::literal_text()>
    literal_text
    ;

qi::rule<Iterator, ast::variable(), qi::space_type>
    variable
    ;

qi::rule<Iterator, ast::variable(), qi::space_type>
    variable_unescaped
    ;
```

stache Parser

```
qi::rule< Iterator, ast::section(), qi::locals<std::string>  
    , qi::space_type>  
    section  
    ;  
  
qi::rule<Iterator, ast::identifier(), qi::space_type>  
    section_begin  
    ;  
  
qi::rule<Iterator, qi::unused_type(std::string), qi::space_type>  
    section_end  
    ;  
  
qi::rule<Iterator, ast::partial(), qi::space_type>  
    partial  
    ;
```

stache Parser

```
node_list =  
    *stache_node  
    ;
```

```
stache_node =  
    no_skip[literal_text]  
    | comment  
    | variable  
    | variable_unescaped  
    | section  
    | partial  
    ;
```

stache Parser

```
node_list =  
    *stache_node  
    ;
```

```
stache_node =  
    no_skip[literal_text]  
    | comment  
    | variable  
    | variable_unescaped  
    | section  
    | partial  
    ;
```

stache Parser

```
literal_text =  
    +(char_ - "{")  
    ;
```

```
stache_node =  
    no_skip[literal_text]  
    | comment  
    | variable  
    | variable_unescaped  
    | section  
    | partial  
    ;
```

stache Parser

```
comment =  
    lit("{ {")  
>> '!''  
>> omit[* (char_ - "}}")]  
>> "}}"  
;
```

```
stache_node =  
    no_skip[literal_text]  
    | comment  
    | variable  
    | variable_unescaped  
    | section  
    | partial  
;
```

stache Parser

```
identifier =  
    lexeme[alpha >> *(alnum | char_('_'))]  
    ;  
  
variable =  
    lit("{ {")  
    >> matches['&']  
    >> identifier  
    >> "}}"  
    ;
```

```
stache_node =  
    no_skip[literal_text]  
    | comment  
    | variable  
    | variable_unescaped  
    | section  
    | partial  
    ;
```


stache Parser

```
identifier =  
    lexeme[alpha >> *(alnum | char_('_'))]  
    ;  
  
variable =  
    lit("{ {")  
    >> matches['&']  
    >> identifier  
    >> "}}"  
    ;
```

```
stache_node =  
    no_skip[literal_text]  
    | comment  
    | variable  
    | variable_unescaped  
    | section  
    | partial  
    ;
```

stache Parser

```
section %=
    matches[&(lit("{") >> '^')]
    >> section_begin[_a = _1]
    >> *stache_node
    >> section_end(_a)
;
```

```
section_begin =
    lit("{")
    >> (lit('#') | '^')
    >> identifier
    >> "}"
;
```

```
section_end =
    lit("{")
    >> '/'
    >> lit(_r1)
    >> "}"
;
```

stache Parser

```
section %=
    matches[&(lit("{{" ) >> '^')]
    >> section_begin[_a = _1]
    >> *stache_node
    >> section_end(_a)
    ;
```

```
section_begin =
    lit("{{"
    >> (lit('#') | '^')
    >> identifier
    >> "}}"
    ;
```

```
section_end =
    lit("{{")
    >> '/'
    >> lit(_r1)
    >> "}}"
    ;
```

stache Parser

```
section %=
    matches[&(lit("{{" ) >> '^')]
    >> section_begin[_a = _1]
    >> *stache_node
    >> section_end(_a)
    ;
```

```
section_begin =
    lit("{{"
    >> (lit('#') | '^')
    >> identifier
    >> "}}"
    ;
```

```
section_end =
    lit("{{"
    >> '/'
    >> lit(_r1)
    >> "}}"
    ;
```

stache AST

```
struct node : boost::spirit::extended_variant<
    undefined
    , comment
    , literal_text
    , variable
    , boost::recursive_wrapper<section>
    , partial
>
{
    node() : base_type() {}
    node(comment const & rhs) : base_type(rhs) {}
    node(literal_text const & rhs) : base_type(rhs) {}
    node(variable const & rhs) : base_type(rhs) {}
    node(section const & rhs) : base_type(rhs) {}
    node(partial const & rhs) : base_type(rhs) {}
};

struct node_list : std::vector<node> {};
```

stache AST

```
struct undefined {};  
  
struct comment {};  
  
struct identifier : std::string  
{  
};  
  
struct literal_text : std::string  
{  
};  
  
struct variable  
{  
    bool is_unescaped;  
    identifier value;  
};  
  
struct partial : identifier  
{  
};  
  
struct section  
{  
    bool is_inverted;  
    identifier name;  
    node_list nodes;  
};
```

stache Printer

```
inline void print(std::ostream& out, node_list const& nodes)
{
    detail::printer p(out);
    for(auto const & node : nodes)
    {
        boost::apply_visitor(p, node);
    }
}
```

stache Printer

```
class printer
{
public:
    typedef void result_type;

    printer(std::ostream& out)
        : out(out)
    {}

    void operator()(undefined) const
    {
        out << "WHOA! we have an undefined" << std::endl;
    }

    void operator()(comment) const
    {
    }

private:
    std::ostream& out;
};
```


stache Printer

```
void operator()(literal_text const & v) const
{
    out << v;
}
```

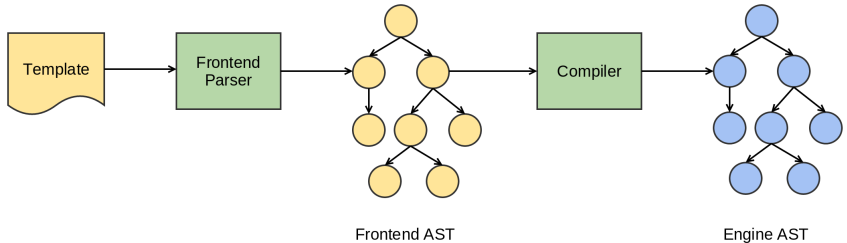
```
void operator()(variable const & v) const
{
    out << "{{";
    if(v.is_unescaped)
    {
        out << "&";
    }
    out << v.value << "}}";
}
```

```
void operator()(section const & v) const
{
    out << "{{";
    if(v.is_inverted) { out << "^"; }
    else { out << "#"; }
    out << v.name << "}}";

    for(auto const & node : v.nodes)
    {
        boost::apply_visitor(*this, node);
    }

    out << "{/" << v.name << "}}";
}
```

The Frontend



Starting Out - load_template

```
template <typename Format, typename Iterator>
inline vm::ast::node load_template( Iterator & begin
                                   , Iterator const & end)
{
    return backend::compile(frontend::parse<Format>(begin,end));
}

template <typename Format>
inline vm::ast::node load_template(std::istream & input)
{
    return backend::compile(frontend::parse<Format>(input));
}
```

stache Compile

```
namespace boost { namespace boostache { namespace backend
{
    inline vm::ast::node compile(frontend::stache::ast::root const & ast)
    {
        return stache_compiler::compile(ast);
    }
}}}
```

```
inline vm::ast::node compile(fe::stache::ast::root const & ast)
{
    detail::stache_visit visit;
    return visit(ast);
}
```

stache Compile

```
namespace boost { namespace boostache { namespace backend
{
    inline vm::ast::node compile(frontend::stache::ast::root const & ast)
    {
        return stache_compiler::compile(ast);
    }
}}}
```

```
inline vm::ast::node compile(fe::stache::ast::root const & ast)
{
    detail::stache_visit visit;
    return visit(ast);
}
```

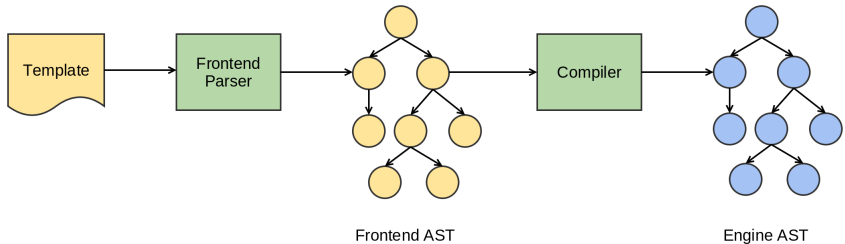
stache Compile - Visitor

```
class stache_visit
{
public:
    using result_type = vm::ast::node;

    vm::ast::node operator() (fe::stache::ast::root const & nodes) const
    {
        vm::ast::node_list node_list;
        for(auto const & node : nodes)
        {
            node_list.nodes.push_back(boost::apply_visitor(*this, node));
        }
        return node_list;
    }

private:
};
```

The Frontend



```
struct node : boost::spirit::extended_variant<
    undefined
    , nop
    , literal
    , variable
    , render
    , boost::recursive_wrapper<for_each>
    , boost::recursive_wrapper<if_then_else>
    , boost::recursive_wrapper<select_context>
    , boost::recursive_wrapper<node_list> >
{

    node() : base_type() {}
    node(nop const & rhs) : base_type(rhs) {}
    node(literal const & rhs) : base_type(rhs) {}
    node(variable const & rhs) : base_type(rhs) {}
    node(render const & rhs) : base_type(rhs) {}
    node(for_each const & rhs) : base_type(rhs) {}
    node(if_then_else const & rhs) : base_type(rhs) {}
    node(select_context const & rhs) : base_type(rhs) {}
    node(node_list const & rhs) : base_type(rhs) {}

};
```



```
struct node : boost::spirit::extended_variant<
    undefined
    , nop
    , literal
    , variable
    , render
    , boost::recursive_wrapper<for_each>
    , boost::recursive_wrapper<if_then_else>
    , boost::recursive_wrapper<select_context>
    , boost::recursive_wrapper<node_list> >
{

    node() : base_type() {}
    node(nop const & rhs) : base_type(rhs) {}
    node(literal const & rhs) : base_type(rhs) {}
    node(variable const & rhs) : base_type(rhs) {}
    node(render const & rhs) : base_type(rhs) {}
    node(for_each const & rhs) : base_type(rhs) {}
    node(if_then_else const & rhs) : base_type(rhs) {}
    node(select_context const & rhs) : base_type(rhs) {}
    node(node_list const & rhs) : base_type(rhs) {}

};
```

```
struct literal
{
    literal(){}
    literal(std::string const & v) : value(v) {}
    std::string value;
};

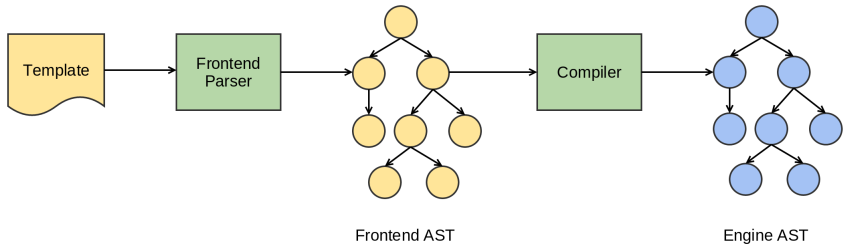
struct for_each
{
    std::string name;
    node value;
};
```

```
struct if_then_else
{
    condition condition_;
    node then_;
    node else_;
};

struct select_context
{
    std::string tag;
    node body;
};

struct node_list
{
    std::vector<node> nodes;
};
```

The Frontend



stache Compiler

```
class stache_visit
{
public:
    using result_type = vm::ast::node;

    vm::ast::node operator() (fe::stache::ast::undefined) const
    {
        return vm::ast::node{};
    }

    vm::ast::node operator() (fe::stache::ast::literal_text const & v) const
    {
        return vm::ast::literal{v};
    }

    vm::ast::node operator() (fe::stache::ast::variable const & v) const
    {
        return vm::ast::render{v.value};
    }

    vm::ast::node operator() (fe::stache::ast::comment const & v) const
    {
        return vm::ast::nop{};
    }

    vm::ast::node operator() (fe::stache::ast::partial const & v) const
    {
        // TODO: need to implement partials
        return vm::ast::nop{};
    }
}
```

stache Compiler - Section

```
vm::ast::node operator() (fe::stache::ast::node_list const & nodes) const
{
    vm::ast::node_list node_list;
    for(auto const & node : nodes)
    {
        node_list.nodes.push_back(boost::apply_visitor(*this, node));
    }
    return node_list;
}
```

stache Compiler - Section

```
vm::ast::node operator() (fe::stache::ast::section const & sec) const
{
    vm::ast::node_list vm_ast;
    for(auto const & node : sec.nodes)
    {
        vm_ast.nodes.push_back(boost::apply_visitor(*this, node));
    }

    vm::ast::for_each section_body;
    section_body.name = sec.name;
    section_body.value = vm_ast;

    vm::ast::if_then_else if_block;
    if_block.condition_.name = sec.name;

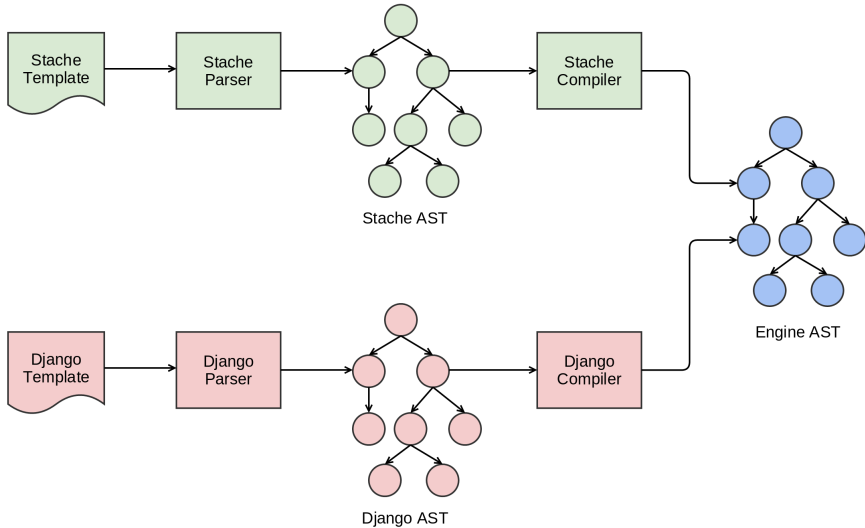
    vm::ast::select_context select;
    select.tag = sec.name;
    select.body = section_body;

    if(sec.is_inverted) { if_block.else_ = select; }
    else                { if_block.then_ = select; }

    return if_block;
}
```

Frontend AST \rightarrow Engine/VM AST

Multiple Frontends



Sample Django Template

```
std::string input(  
    "My name is {{another.name}}. "  
    "{# This is a comment #}"  
    "I am {{pet}} years old."  
    "{% if another.notok %}"  
    "Nope"  
    "{% else %}"  
    "Yep"  
    "{% endif %}\n");
```

Django dot notation

```
vm::ast::node operator() (fe::django::ast::variable const & v) const
{
    vm::ast::node body = vm::ast::render{v.back()};
    for(auto iter = --v.rend(); iter != v.rbegin(); --iter)
    {
        vm::ast::select_context select;
        select.tag = *iter;
        select.body = std::move(body);
        body = std::move(select);
    }
    return body;
}
```

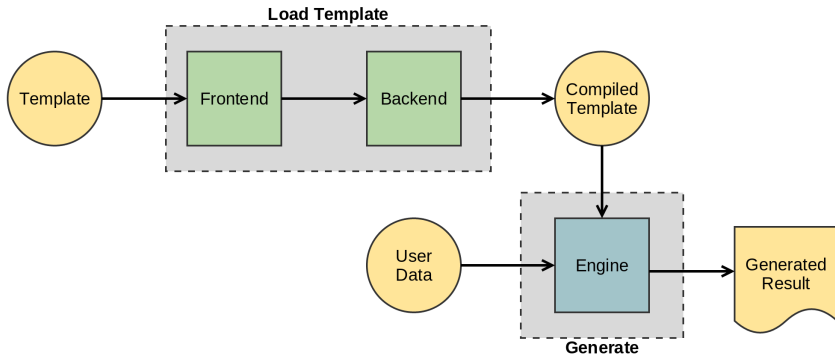
Django If

```
vm::ast::node operator() (
    fe::django::ast::if_elif_else const & if_elif_else) const
{
    vm::ast::node_list then_;
    for(auto const & node : if_elif_else.if_.body)
    {
        then_.nodes.push_back(boost::apply_visitor(*this, node));
    }

    vm::ast::if_then_else if_then_else;
    if_then_else.condition_.name = if_elif_else.if_.condition.front();
    if_then_else.then_ = std::move(then_);
    if(static_cast<bool>(if_elif_else.else_))
    {
        vm::ast::node_list else_;
        for(auto const & node : if_elif_else.else_.get())
        {
            else_.nodes.push_back(boost::apply_visitor(*this, node));
        }
        if_then_else.else_ = std::move(else_);
    }

    return if_then_else;
}
```

The Big Picture



generate

```
using boostache::load_template;
using boostache::format::stache;

auto iter = input.begin();
auto templ = load_template<stache>(iter, input.end());

std::stringstream stream;
boostache::generate(stream, templ, invoice);
```

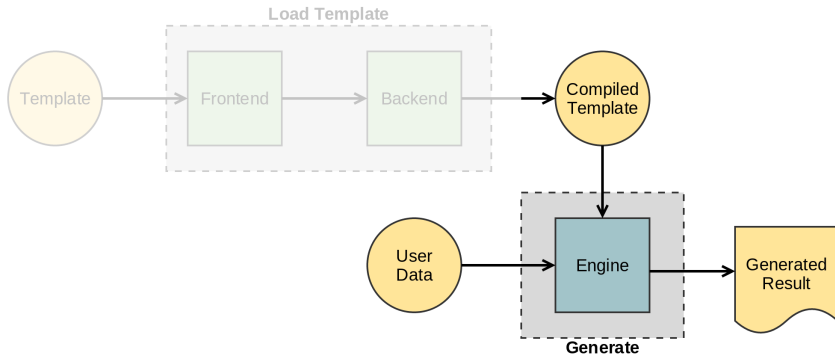
generate

```
using boostache::load_template;
using boostache::format::stache;

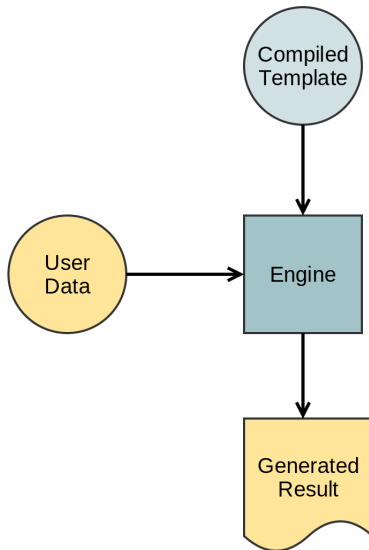
auto iter = input.begin();
auto templ = load_template<stache>(iter, input.end());

std::stringstream stream;
boostache::generate(stream, templ, invoice);
```

The Engine / VM



The Engine



generate

```
template <typename Stream, typename Context>
void generate( Stream & stream
              , vm::ast::node const & templ
              , Context const & context)
{
    vm::generate(stream, templ, context);
}
```

Detail generate call:

```
template <typename Stream, typename Template, typename Context>
void generate( Stream & stream
              , Template const & templ
              , Context const & ctx)
{
    engine_visitor_base<Stream, Context> engine(stream, ctx);
    engine(templ);
}
```

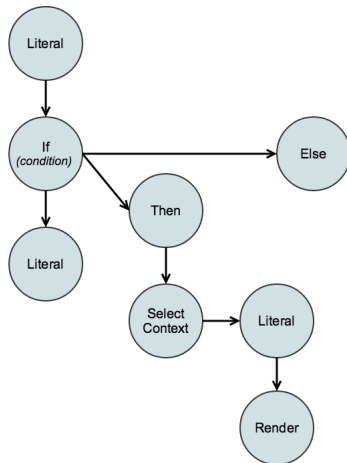
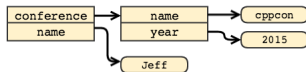
generate

```
template <typename Stream, typename Context>
void generate( Stream & stream
              , vm::ast::node const & templ
              , Context const & context)
{
    vm::generate(stream, templ, context);
}
```

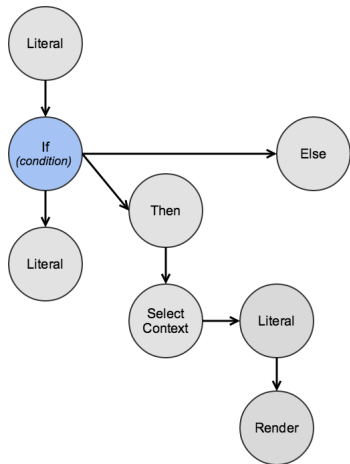
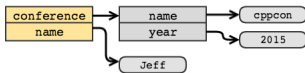
Detail generate call:

```
template <typename Stream, typename Template, typename Context>
void generate( Stream & stream
              , Template const & templ
              , Context const & ctx)
{
    engine_visitor_base<Stream, Context> engine(stream, ctx);
    engine(templ);
}
```

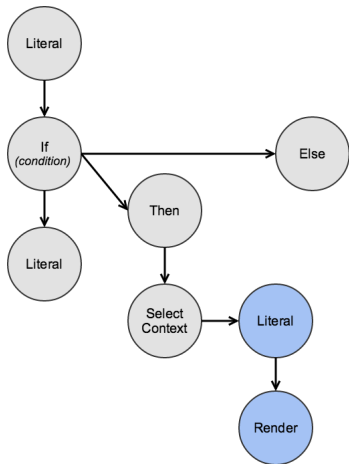
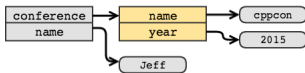
VM Processing



VM Processing



VM Processing



engine_visitor

```
template <typename Stream, typename Context>
class engine_visitor_base
{
public:
    using result_type = void;

    engine_visitor_base(Stream & s, Context const & c)
        : stream(s)
        , context(c)
    {}

    void operator()(ast::node_list const & nodes) const
    {
        for(auto const & node : nodes.nodes)
        {
            boost::apply_visitor(*this, node);
        }
    }

    void operator()(ast::node const & node) const
    {
        boost::apply_visitor(*this, node);
    }

private:
    Stream & stream;
    Context const & context;
};
```

engine_visitor

```
void operator() (ast::undefined) const  
{}
```

```
void operator() (ast::nop) const  
{}
```

```
void operator() (ast::literal const & lit) const  
{  
    using boost::boostache::extension::render;  
    render(stream, lit.value);  
}
```

```
void operator() (ast::render const & r) const  
{  
    using boost::boostache::extension::render;  
    render(stream, context, r.name);  
}
```


engine_visitor

```
void operator() (ast::if_then_else const & v) const
{
    using boost::boostache::extension::test;
    if(test(context, v.condition_.name))
    {
        boost::apply_visitor(*this, v.then_);
    }
    else
    {
        boost::apply_visitor(*this, v.else_);
    }
}
```

```
void operator() (ast::for_each const & v) const
{
    using boost::boostache::vm::detail::foreach;
    foreach(stream, v, context);
}
```

foreach magic

```
template <typename Stream, typename Node, typename Context>
void foreach(Stream & stream, Node const & node, Context const & context)
{
    using boostache::vm::detail::foreach;
    foreach( stream
        , node
        , context
        , extension::foreach_category_t<Context>{});
}
```

foreach magic

```
namespace boost { namespace boostache { namespace extension
{
    // -----
    // classifications
    struct category_attribute {};
    struct unused_attribute : category_attribute {};
    struct plain_attribute : category_attribute {};
    struct sequence_attribute : category_attribute {};
    struct associative_attribute : category_attribute {};
    struct tuple_attribute : category_attribute {};
    struct variant_attribute : category_attribute {};
    struct optional_attribute : category_attribute {};
}}}
```

foreach magic

```
template <typename T, typename Enable = void>
struct foreach_category
    : mpl::identity<plain_attribute> {};
```

```
template <>
struct foreach_category<std::string>
    : mpl::identity<plain_attribute> {};
```

```
template <typename T>
struct foreach_category< T
                        , vm::trait::enable_if_is_variant_t<T>
                        >
    : mpl::identity<variant_attribute> {};
```

```
template <typename T>
struct foreach_category< T
                        , vm::trait::enable_if_sequence_not_map_t<T>
                        >
    : mpl::identity<sequence_attribute> {};
```

foreach magic

```
template <typename T>  
struct foreach_category<boost::optional<T>>  
    : mpl::identity<optional_attribute> {};
```

```
template <typename T>  
struct foreach_category<std::map<std::string,T>>  
    : mpl::identity<associative_attribute> {};
```

```
template <typename T>  
using foreach_catetgory_t = typename foreach_category<T>::type;
```

foreach magic

```
template < typename Stream, typename Node, typename Context
          , typename Category>
void foreach( Stream & stream
             , Node const & node
             , Context const & context
             , Category)
{
    generate(stream, node.value, context);
}
```

foreach magic

```
template <typename Stream, typename Node>
struct unwrap_variant_foreach
{
    using result_type = void;

    unwrap_variant_foreach(Stream & stream, Node const & node)
        : stream_(stream), node_(node)
    {}

    template <typename T>
    void operator()(T const & context) const
    {
        vm::detail::foreach(stream_, node_, context);
    }

    Stream & stream_;
    Node const & node_;
};

template <typename Stream, typename Node, typename Context>
void foreach( Stream & stream
            , Node const & node
            , Context const & context
            , extension::variant_attribute)
{
    extension::detail::unwrap_variant_foreach<Stream, Node>
        variant_foreach(stream, node);
    boost::apply_visitor(variant_foreach, context);
}
```


foreach magic

```
template <typename Stream, typename Node>
struct unwrap_variant_foreach
{
    using result_type = void;

    unwrap_variant_foreach(Stream & stream, Node const & node)
        : stream_(stream), node_(node)
    {}

    template <typename T>
    void operator()(T const & context) const
    {
        vm::detail::foreach(stream_, node_, context);
    }

    Stream & stream_;
    Node const & node_;
};

template <typename Stream, typename Node, typename Context>
void foreach( Stream & stream
            , Node const & node
            , Context const & context
            , extension::variant_attribute)
{
    extension::detail::unwrap_variant_foreach<Stream, Node>
        variant_foreach(stream, node);
    boost::apply_visitor(variant_foreach, context);
}
```

foreach magic

```
template <typename Stream, typename Node, typename Context>
void foreach( Stream & stream
             , Node const & node
             , Context const & ctx
             , extension::optional_attribute)
{
    if(ctx)
    {
        foreach( stream, node, *ctx
                , extension::foreach_category_t<decltype(*ctx)>{});
    }
    else
    {
        generate(stream, node.value, ctx);
    }
}
```

foreach magic

```
template <typename Stream, typename Node, typename Context>
void foreach( Stream & stream
             , Node const & node
             , Context const & context
             , extension::sequence_attribute)
{
    for(auto const & item : context)
    {
        generate(stream, node.value, item);
    }
}
```

Select Context Magic

Back to the `engine_visitor_base`

```
void operator() (ast::select_context const & select_ctx) const
{
    select_context_dispatch( stream, select_ctx, context
                           , extension::select_category_t<Context>{} );
}
```

Select Context Magic

```
template <typename Stream, typename Context, typename Category>
void select_context_dispatch( Stream & stream
                             , ast::select_context const & templ
                             , Context const & ctx, Category)
{
    generate(stream, templ.body, ctx);
}
```

Select Context Magic

```
template <typename Stream, typename Context>
void select_context_dispatch( Stream & stream
                             , ast::select_context const & templ
                             , Context const & ctx
                             , extension::associative_attribute)
{
    auto iter = ctx.find(templ.tag);
    if(iter != ctx.end())
    {
        select_context( stream, templ.body, ctx, iter->second
                        , extension::select_category_t<decltype(iter->second)>{});
    }
    else
    {
        generate(stream, templ.body, ctx);
    }
}
```

Select Context Magic

```
template < typename Stream, typename Template
        , typename Context1, typename Context2
        , typename CategoryChild
        >
void select_context( Stream & stream, Template const & templ
                   , Context1 const & ctx_parent
                   , Context2 const & /*ctx_child*/
                   , CategoryChild)
{
    generate(stream, templ, ctx_parent);
}
```

Select Context Magic

```
template < typename Stream, typename Template
        , typename Context1, typename Context2
        >
void select_context( Stream & stream, Template const & templ
                    , Context1 const & /*ctx_parent*/
                    , Context2 const & ctx_child
                    , extension::associative_attribute)
{
    generate(stream, templ, ctx_child);
}
```

```
template < typename Stream, typename Template
        , typename Context1, typename Context2
        >
void select_context( Stream & stream, Template const & templ
                    , Context1 const & /*ctx_parent*/
                    , Context2 const & ctx_child
                    , extension::sequence_attribute)
{
    generate(stream, templ, ctx_child);
}
```


Select Context Magic

```
template < typename Stream, typename Template
        , typename Context1, typename Context2
        >
void select_context( Stream & stream, Template const & templ
                   , Context1 const & ctx_parent
                   , Context2 const & ctx_child
                   , extension::variant_attribute)
{
    boost::apply_visitor(
        unwrap_and_select_context< Stream
                                , Template
                                , Context1>{ stream
                                           , templ
                                           , ctx_parent}
        , ctx_child
    );
}
```

Select Context Magic

```
template < typename Stream, typename Template
        , typename Context1, typename Context2
        >
void select_context( Stream & stream, Template const & templ
                    , Context1 const & ctx_parent
                    , Context2 const & ctx_child
                    , extension::variant_attribute)
{
    boost::apply_visitor(
        unwrap_and_select_context< Stream
                                , Template
                                , Context1>{ stream
                                            , templ
                                            , ctx_parent}
        , ctx_child
    );
}
```

Customization Point

```
std::string input(  
    "LiaW subject topic is: \n"  
    "{{# secret}}Not telling{{/ secret}}"  
    "{{^ secret}}{{name}}{{/ secret}}"  
);  
  
std::function<bool()> conf_started =  
    [](){return true;};  
  
std::function<std::string()> liaw_topic =  
    [](){return "Jeff's latest issue";};  
  
smodel_t data = {  
    {"name" , liaw_topic},  
    {"secret" , conf_started}  
};
```

Customization Point

```
namespace boost { namespace boostache { namespace extension
{
    template <typename T>
    bool test( std::string const & name
              , std::function<T()> const & context
              , extn::plain_attribute)
    {
        return test(name, context());
    }

    template< typename Stream
              , typename T
              >
    void render( Stream & stream
               , std::function<T()> const & context
               , std::string const & name
               , extn::plain_attribute)
    {
        render(stream, context(), name);
    }
}}}
```

Part III

Future

What is next?

- ▶ General clean-up
- ▶ Complete Mustache and Django support
- ▶ Complete generalizing category handling
- ▶ Flesh out the extension mechanism
- ▶ Docs, docs, docs

Where do I find it?

`https://github.com/cierelabs/boostache`



Part IV

Bonus

Fusion

```
template <typename Seq>
int tag_index(std::string const & tag)
{
    return
        detail::tag_index
            < Seq
            , fusion::extension::struct_size<Seq>::value-1
            >::call(tag);
}
```

```
template <typename Seq>
bool has_tag(std::string const & tag)
{
    return tag_index<Seq>(tag) >= 0;
}
```

Fusion

```
template <typename Seq, int I>
struct tag_index
{
    static int call(std::string const & tag)
    {
        if(fusion::extension::struct_member_name<Seq, I>::call()
            == tag)
        {
            return I;
        }
        else
        {
            return tag_index<Seq, I-1>::call(tag);
        }
    }
};
```

Fusion

```
template <typename Seq>
struct tag_index<Seq, 0>
{
    static int call(std::string const & tag)
    {
        if(fusion::extension::struct_member_name<Seq, 0>::call()
            == tag)
        {
            return 0;
        }
        else
        {
            return -1;
        }
    }
};
```

Fusion

```
template <typename S>
void render_tag(S & s, std::string const & tag)
{
    detail::render_tag
        < S
        , fusion::extension::struct_size<S>::value-1
        >::call(s, tag);
}
```

Fusion

```
template <typename S, int I>
struct render_tag
{
    static void call(S & s, std::string const & tag)
    {
        if(fusion::extension::struct_member_name<S,I>::call()
            == tag)
        {
            render<S,I>(s);
        }
        else
        {
            return render_tag<S,I-1>::call(s, tag);
        }
    }
};
```

Fusion

```
template <typename S>
struct render_tag<S,0>
{
    static void call(S & s, std::string const & tag)
    {
        if(fusion::extension::struct_member_name<S,0>::call()
            == tag)
        {
            render<S,0>(s);
        }
    }
};
```

Fusion

```
template <typename S, int I>
void render(S & s)
{
    std::cout
        << boost::fusion::extension::
            access::struct_member<S,I>::
                template apply<S> :: call(s)
        ;
};
```