Yandex

Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

Piotr Reznikov, Sergei Khandrikov

The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

Who We Are

Hello, we are Yandex

Yandex is the leading search engine in Russia.

- 62% of Russian search traffic
- 25 mln unique users per day
- 6,000+ employees

We do mail, as well

> Yandex.Mail is a free mail service, quite popular in Russia and Russian-speaking countries.

- Built in 2000
- 9 million unique users per day
- 110 million messages sent and received daily

The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

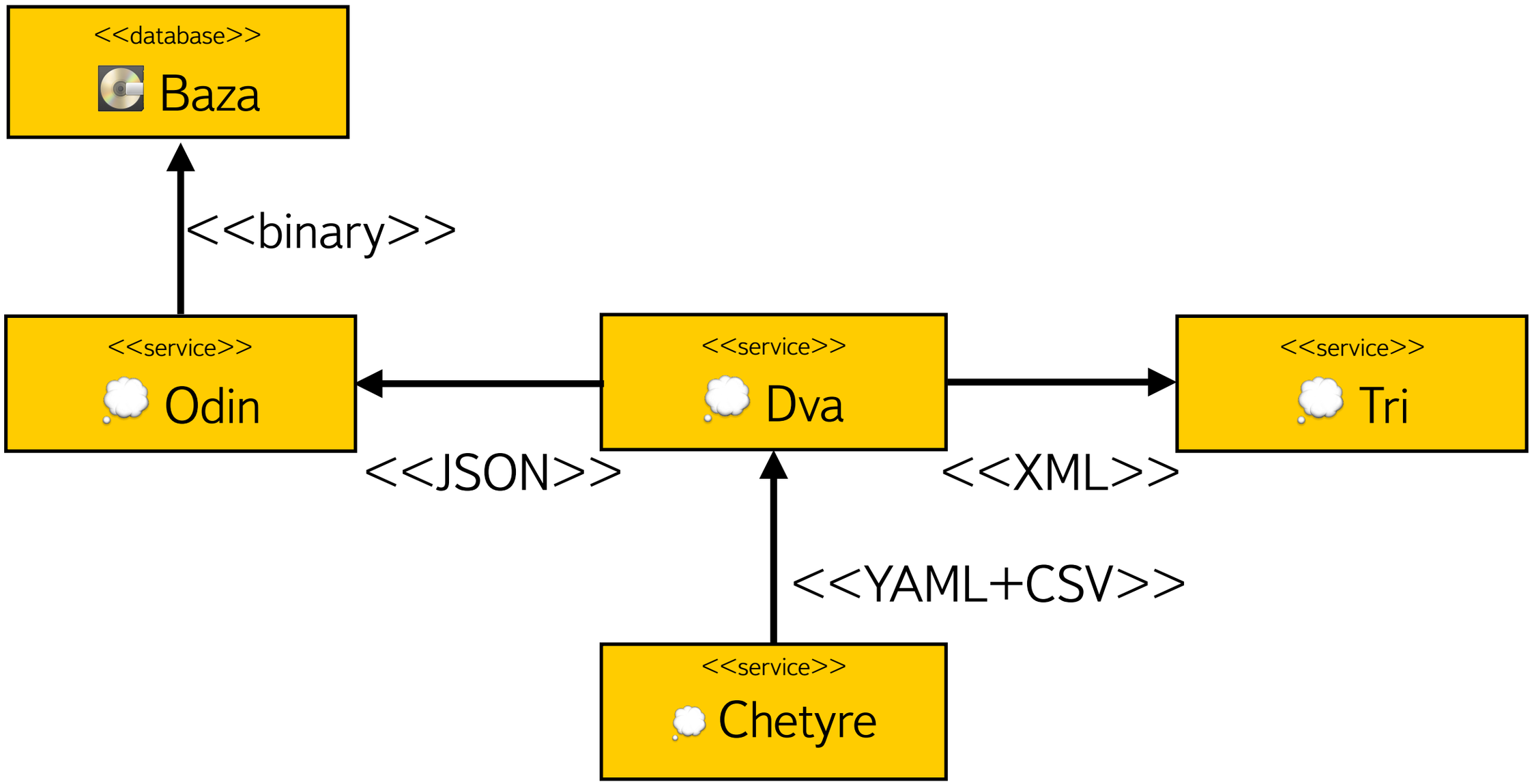
Problem

Situation

> There are dozens of services inside and outside Mail which are:

- written in C++
- communicating via HTTP protocol with different text formats
- getting a data in binary format from a database

Situation


<<database>>
Baza
<<binary>>
<<service>>
Odin
<<service>>
Dva
<<service>>
Tri
<<JSON>>
<<XML>>
<<YAML+CSV>>
<<service>>
Chetyre

Situation

So, for various formats, we want

- Unification - convenient mechanism for C++.
- Optimisation - maximum performance.

We need to serialize some struct

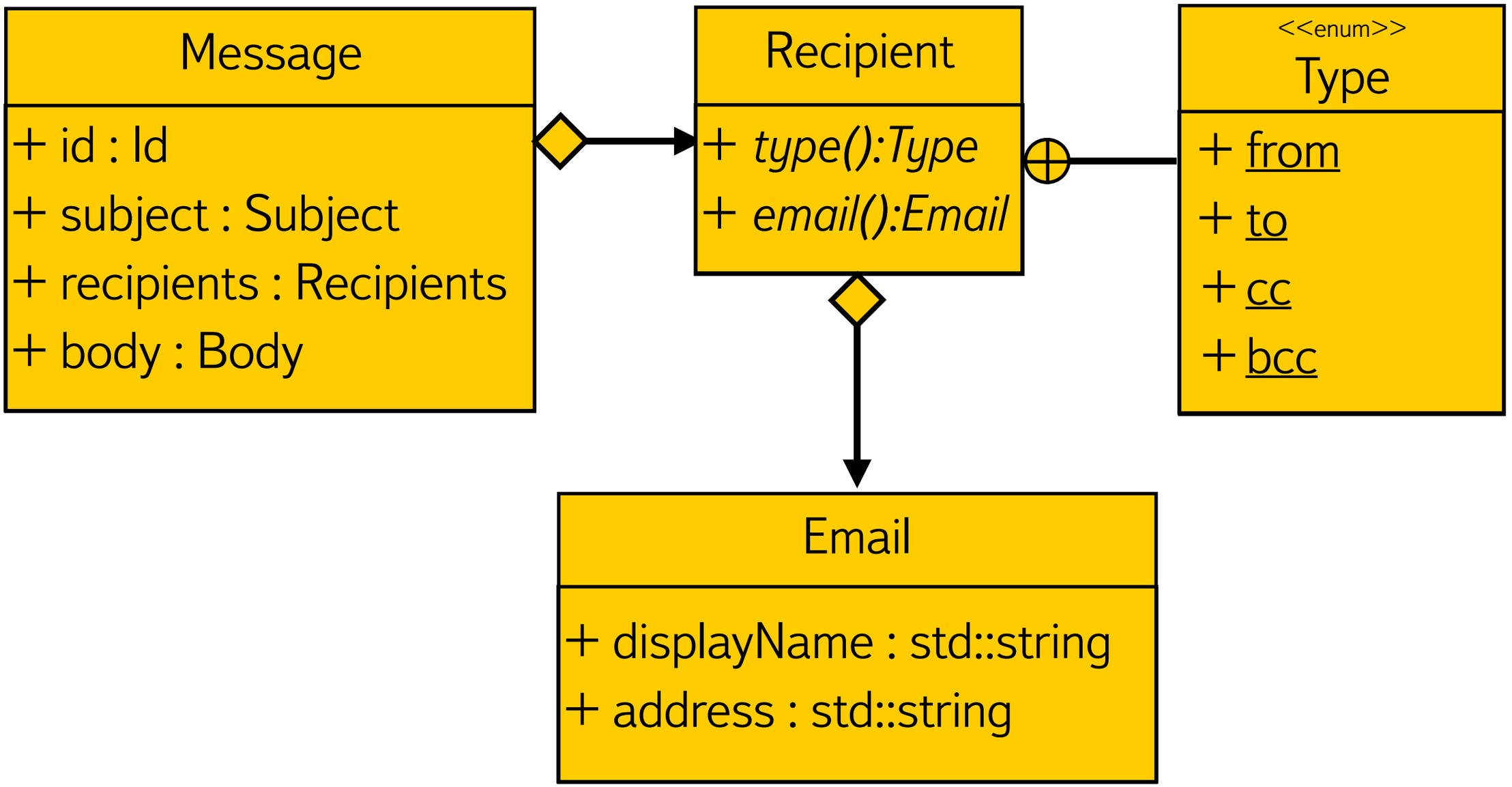
> Here's some trivialized e-mail class system:

```
struct Message {
    std::string id;
    std::string subject;
    std::string body;
    std::vector<Recipient> recipients;
};
```

```
struct Recipient {
    enum class Type { … };
    Type type() const;
    Email email() const;
};

struct Email {
    std::string name;
    std::string address;
};
```

We need to serialize some struct



Serialize into JSON

> So let’s just put some YAJL here, right?

Serialize into JSON

So let’s just put some YAJL here, right?

```
#define YAJL_ADD_FIELD_NAME(gen, name) \
    const unsigned char * const name##Name = reinterpret_cast<const unsigned cha
    yajl_gen_string(gen, name##Name, strlen(#name));

#define YAJL_ADD_STRING(gen, str) \
    yajl_gen_string(gen, reinterpret_cast<const unsigned char*>(str.c_str()), st

#define YAJL_ADD_FIELD(gen, obj, field) \
    YAJL_ADD_FIELD_NAME(gen, field) \
    YAJL_ADD_STRING(gen, obj.field)


void print(yajl_gen_t* gen, const model::Email& e) {
    yajl_gen_map_open(gen);
    YAJL_ADD_FIELD(gen, e, name);
    YAJL_ADD_FIELD(gen, e, address);
    yajl_gen_map_close(gen);
}

void print(yajl_gen_t* gen, const model::Recipient& r) {
    yajl_gen_map_open(gen);
    YAJL_ADD_FIELD_NAME(gen, type);
    YAJL_ADD_STRING(gen, r.type());
    YAJL_ADD_FIELD_NAME(gen, email);
    print(gen, r.email());
    yajl_gen_map_close(gen);
}

void print(yajl_gen_t* gen, const model::Message& m) {
    yajl_gen_map_open(gen);

    YAJL_ADD_FIELD(gen, m, id);
    YAJL_ADD_FIELD(gen, m, body);
    YAJL_ADD_FIELD(gen, m, subject);

    YAJL_ADD_FIELD_NAME(gen, recipients);

    yajl_gen_array_open(gen);
    for( const auto& r : m.recipients ) {
        print(gen, r);
    }
    yajl_gen_array_close(gen);

    yajl_gen_map_close(gen);
}
```

```
std::string toJson(Message m) {
  yajl_gen_t* gen = yajl_gen_alloc(nullptr);
  yajl_gen_map_open(gen);
  YAJL_ADD_FIELD(gen, m, id); //we try our
best to make that code look nice
  YAJL_ADD_FIELD(gen, m, body);
  YAJL_ADD_FIELD(gen, m, subject);
  YAJL_ADD_FIELD_NAME(gen, recipients);
  yajl_gen_array_open(gen);
  for( const auto& r : m.recipients ) {
    //...
  }
```

Serialize into JSON and XML

> But hey, you are only need to write this once. Or are you, really? How about XML?

Deserialize from database format

> Another hundreds lines of code?

Serialize without «body» field

> Exactly like before, but without heavy «body» field

Serialize without «body» field

> Which solution do you prefer?

- `std::string toJson(Message m, bool useBody); //non-extensible`
- `std::string toJson(Message m); //fat copy-paste`
  `std::string toJsonNoBody(Message m);`
- `std::string toJson(Message m); //DTO, copy-paste`
  `std::string toJson(MessageNoBody m);`

> We like neither.

Another caveats in handwritten solution

- Repeat for every struct you want to serialize
- Inverse-repeat for deserialize (for every struct, for every format)

We want pretty solution like this

```
JsonWriter jWriter;

Message msg;
auto json = jWriter.apply(msg);
```

```
template<typename T>
string JsonWriter::apply(T t);
```

We want pretty solution like this

```
DBReader dbReader;     //deserializator for binary DB protocol
JsonWriter jWriter;

auto data = db.getMessages();

auto msg = dbReader.apply<Message>(data); //parse binary format
msg.adjustRecipients();                   //make adjustments to object
auto json = jWriter.apply(msg);           //give it away as json
```

We want pretty solution like this

```
JsonWriter jWriter;

Message msg;
auto json = jWriter.apply(msg);

struct AnOther { int x; };
ADAPT_STRUCT(AnOther, (int, x)); //Sad but true: no native
                                 //introspection in C++

auto jsonRec = jWriter.apply(AnOther()); //Same mechanism for any type
```

We want pretty solution like this

```
JsonWriter jWriter;

Message msg;
auto json = jWriter.apply(msg);

ADAPT_VIEW(Message, MessageNoBodyView, ...)

auto jsonNoBody = jWriter.apply<MessageNoBodyView>(msg); //Same type,
                                                         //different view. No copies made
```

The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

🚲 The YReflection

Research subjects for the new solution 🚲

- Easy definition - no external tools
- Native - no Data Transfer Objects
- Documented
- Zero copy support

Basic Idea

> Inspired by the Stack Overflow answer from Seth Heeren (SeHe) [1].
> Walking through the objects whose types are defined with Boost.Fusion

- Define meta information with Boost.Fusion
- Visiting data entities
- Apply appropriate visitor for each attribute or method

We want pretty solution like this

```
JsonWriter jWriter;

Message msg;
auto json = jWriter.apply(msg);
```

Define Metadata With Boost.Fusion

So it might looks like this:

```
struct Message {
    std::string id;
    std::string subject;
    std::string body;
    RecipientList recipients;
};

BOOST_FUSION_ADAPT_STRUCT(Message,
    id,
    subject,
    body,
    recipients
)
```

```
BOOST_FUSION_ADAPT_ADT(
    Recipient,
    (obj.type(), obj.type(val))
    (obj.email(), obj.email(val))
)


BOOST_FUSION_ADAPT_STRUCT(
    Email,
    name,
    address
)
```

Define Metadata With Boost.Fusion

We want do define metadata using the Boost.Fusion Adapted like

- BOOST_FUSION_ADAPT_STRUCT //For existing type
- BOOST_FUSION_ADAPT_ADT //For type with setters/getters
- BOOST_FUSION_DEFINE_STRUCT //To define brand new struct

New struct may be defined without double type of members:

```
BOOST_FUSION_DEFINE_STRUCT(
    Email,
    (std::string, name)
    (std::string, address)
)
```

What if the data type being changed

> We want to use BOOST_FUSION_ADAPT_NAMED:

```
//Lightened version of response
BOOST_FUSION_ADAPT_STRUCT_NAMED( Message, MessageNoBodyView,
    id, subject, recipients
)
```

> JSON:

```
{
  "id": "42-100500",
  "subject": "I love you Ozzy!",
  "recipients": [ ... ]
}
```

The documentation

> We want to add inline documentation looks like this:

```
YREFLECTION_ADAPT_DOC( Message, ‘User mail message.’,
    (id, ‘An unique message id.’)
    (subject, ‘Subject header of a message.’)
    (body, ‘Message text.’)
    (recipients, ‘Message recipients.’) )
```

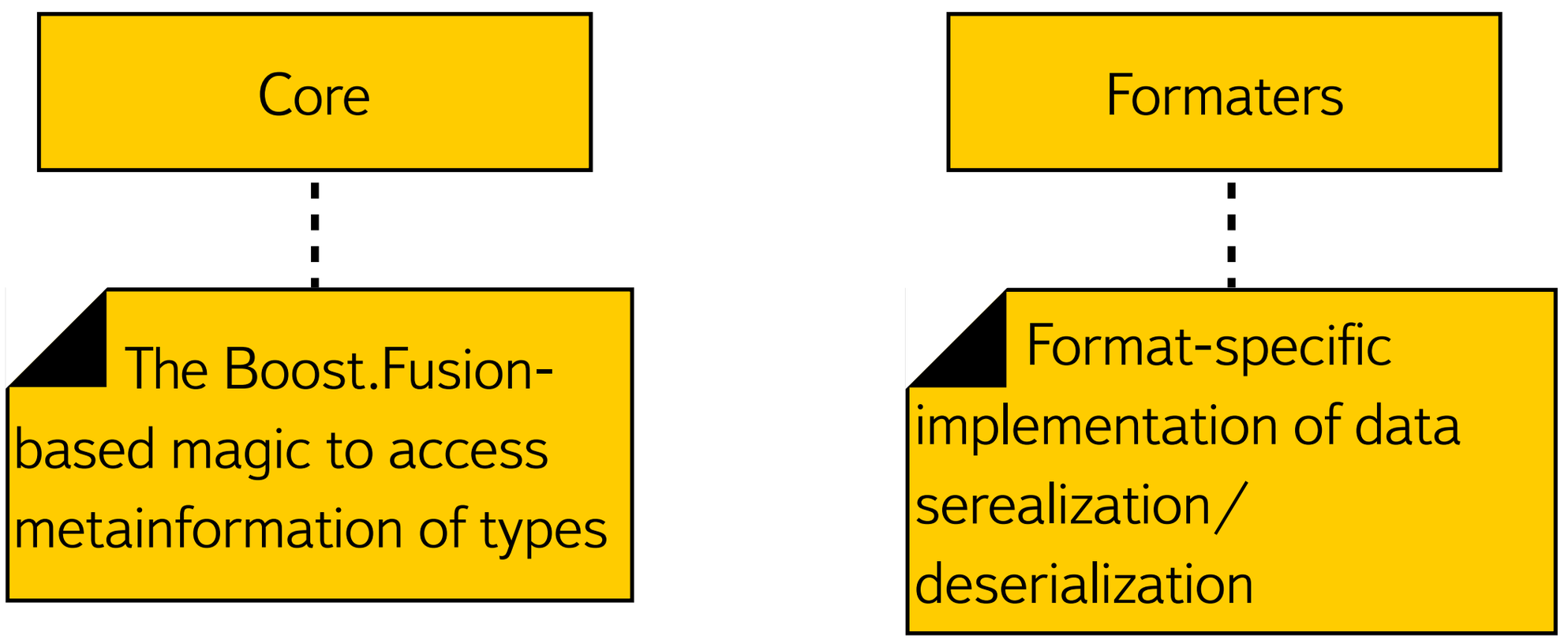
> So we want to get online documentation looks like this:

```
$ curl "http://service/messages/help?format=yaml"
Message               # User mail message.
  id: int             # An unique message id.
  subject: Subject # Subject header of a message.
  body: string        # Message text.
  recipients: vector<Recipient> # Message recipients.
```

The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

Architecture

The Architecture



The Core Magic Example

> When you do this:

```
Message msg;
auto json = JsonWriter().apply(msg);
```

Core Magic Example

When you do this:

```
Message msg;
auto json = JsonWriter().apply(msg);
```

```
string JsonWriter::apply(T t) {
    core::apply(t, *this);
    return yajl.result();
}
```

Core Magic Example

When you do this:

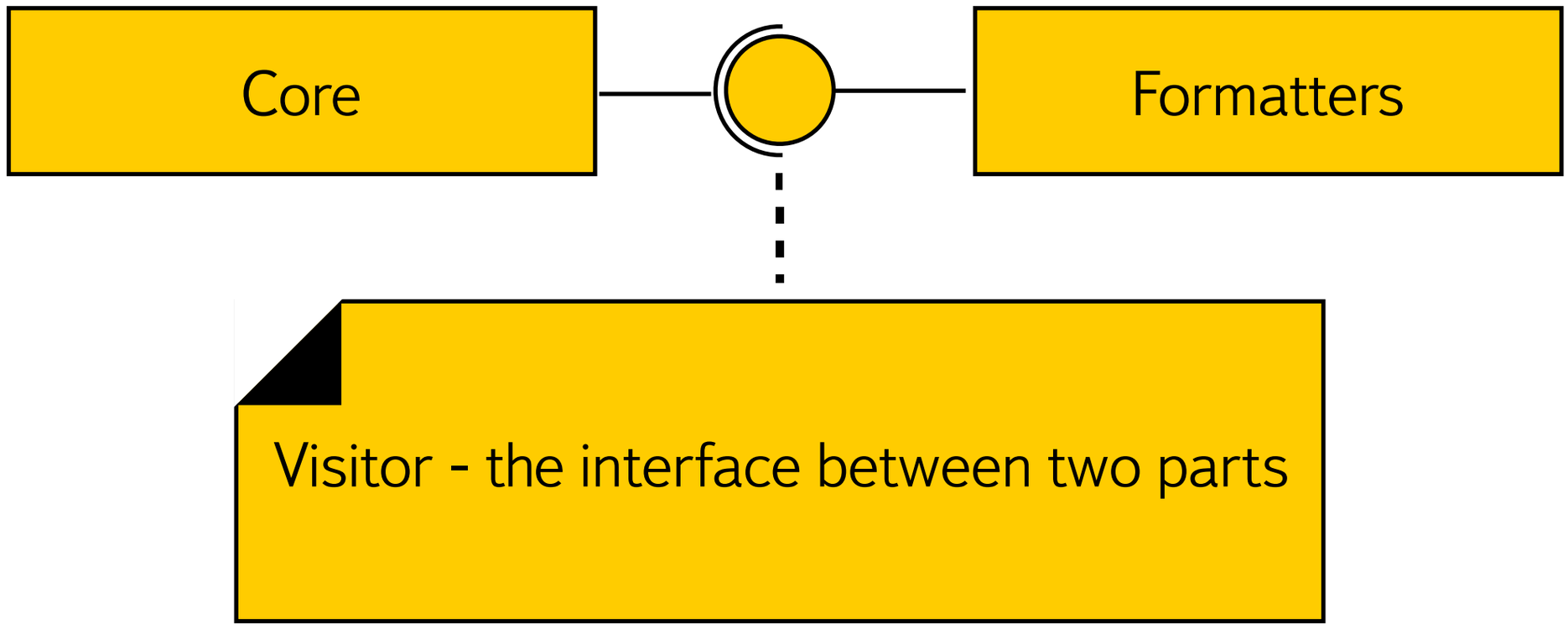
```
Message msg;
auto json = JsonWriter().apply(msg);
```

```
string JsonWriter::apply(T t) {
    core::apply(t, *this);
    return yajl.result();
}
```

The core meets Message structure and uses this code:

```
template <typename Tag>
static void apply (T& field, Visitor& v, Tag tag) {
    auto members = members::make_vector(field);
    auto itemVisitor = v.onStructStart(field, tag);
    boost::fusion::for_each(members, visit_struct::adapt(itemVisitor));
    v.onStructEnd(field, tag);
}
```

Architecture



Visitor


JsonVisitor
+ onField(val : Field&&, tag : Tag)
+ onStructStart(s : Struct&&, tag : Tag) : Visitor
+ onStructEnd(s : Struct&&, tag : Tag)
+ onMapStart(s : Map&&, tag : Tag) : Visitor
+ onMapEnd(s : Map&&, tag : Tag)
+ onSequenceStart(s : Sequence&&, tag : Tag) : Visitor
+ onSequenceEnd(s : Sequence&&, tag : Tag)
+ onOptional(v : Optional&&, tag : Tag) : bool
+ onSmartPointer(v : Pointer&&, tag : Tag) : bool

Visitor

Field handling methods

```
template<typename V, typename Tag> void onField(V&& , Tag);

template<typename V, typename Tag> Visitor onStructStart(V&& , Tag);
template<typename V, typename Tag> void onStructEnd(V&& , Tag);

template<typename V, typename Tag> Visitor onMapStart(V&& , Tag);
template<typename V, typename Tag> void onMapEnd(V&& , Tag);

template<typename V, typename Tag> Visitor onSequenceStart(V&& , Tag);
template<typename V, typename Tag> void onSequenceEnd(V&& , Tag);

template<typename V, typename Tag> bool onOptional(V&& , Tag);

template<typename V, typename Tag> bool onSmartPointer(V&& , Tag);
```

Visitor

Field's tags

```
struct MapItemTag;
struct SequenceItemTag;

template <typename Name>
struct NamedItemTag;

template <typename ... Args>
auto name(const NamedItemTag<Args...>& tag);
```

Visitor

> Plain Field method

```
template<typename V, typename Tag> void onField(V&& , Tag);
```

> Actually a group of specialized method

```
template<typename V, typename ... Arg>
void onField(const V& v, NamedItemTag<Args...> tag) {
    addJsonMapItem(name(tag), boost::lexical_cast<std::string>(v));
}

template<typename V>
void onField(const V& v, SequenceItemTag) {
    addJsonArrayItem(boost::lexical_cast<std::string>(v));
}
```

The Visitor Concept

> Classes handle methods

```
template<typename V, typename Tag> Visitor onStructStart(V&& , Tag);
template<typename V, typename Tag> void onStructEnd(V&& , Tag);
```

```
template<typename V, typename Tag> Visitor onMapStart(V&& , Tag);
template<typename V, typename Tag> void onMapEnd(V&& , Tag);

template<typename V, typename Tag> Visitor onSequenceStart(V&& , Tag);
template<typename V, typename Tag> void onSequenceEnd(V&& , Tag);
```

The Visitor Concept

> Classes handle methods

```
template<typename V, typename ... Args>
Visitor onStructStart(const V& , NamedItemTag<Args...> tag) {
    openJsonMap(name(tag));
    return Visitor(jsonHandle);
}

template<typename V, typename Tag>
void onStructEnd(const V& , Tag) {
    closeJsonMap();
}
```

The Visitor Concept

> Return Visitor allows handle mixed format

```
template<typename V, typename ... Args>
CSVReader JsonReader::onSequenceStart(CSVSequence& ,
                                      NamedItemTag<Args...> tag) {
    auto data = node(name(tag)).data();
    return CSVReader(data);
}

template<typename V, typename Tag>
void JsonReader::onSequenceEnd(const V& , Tag) {
}
```

The Visitor Concept

Optional field handle methods

```
template<typename V, typename Tag> bool onOptional(V&& , Tag);
template<typename V, typename Tag> bool onSmartPointer(V&& , Tag);
```

Trivial implementation for serialization

```
template<typename V, typename Tag> bool onOptional(V const& v, Tag) {
    return v.is_initialized();
}
```

The Visitor Concept

> Trivial implementation for deserialization

```
template<typename V, typename ... Args>
bool onOptional(V& v, NamedItemTag<Args...> tag) {
    if (currentName() == name(tag)) {
        v = V();
        return true;
    }
    return false;
}
```

The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

🏁 Competition

Competitors

〉 *yajl* - hand-made serialization via yajl library and simple http server

〉 *YReflection* - serialization via yreflection and simple http server

〉 *Protobuf* - http server with protobuf support, as a different approach

What do we measure

- Average latency
- CPU consumption
- Memory consumption
- Lines of code (LOC) as a code complexity

Hardware

- Intel® Xeon® Processor E5645 (12M Cache, 2.40 GHz)
- Memory 92 Gb
- Net 2x 1000baseT-FD.

Software

- Ubuntu 14.04.4 x86_64
- gcc version 4.8.4
- Boost 1.60
- Fusion 2.2
- yajl 2
- Protobuf 2.5

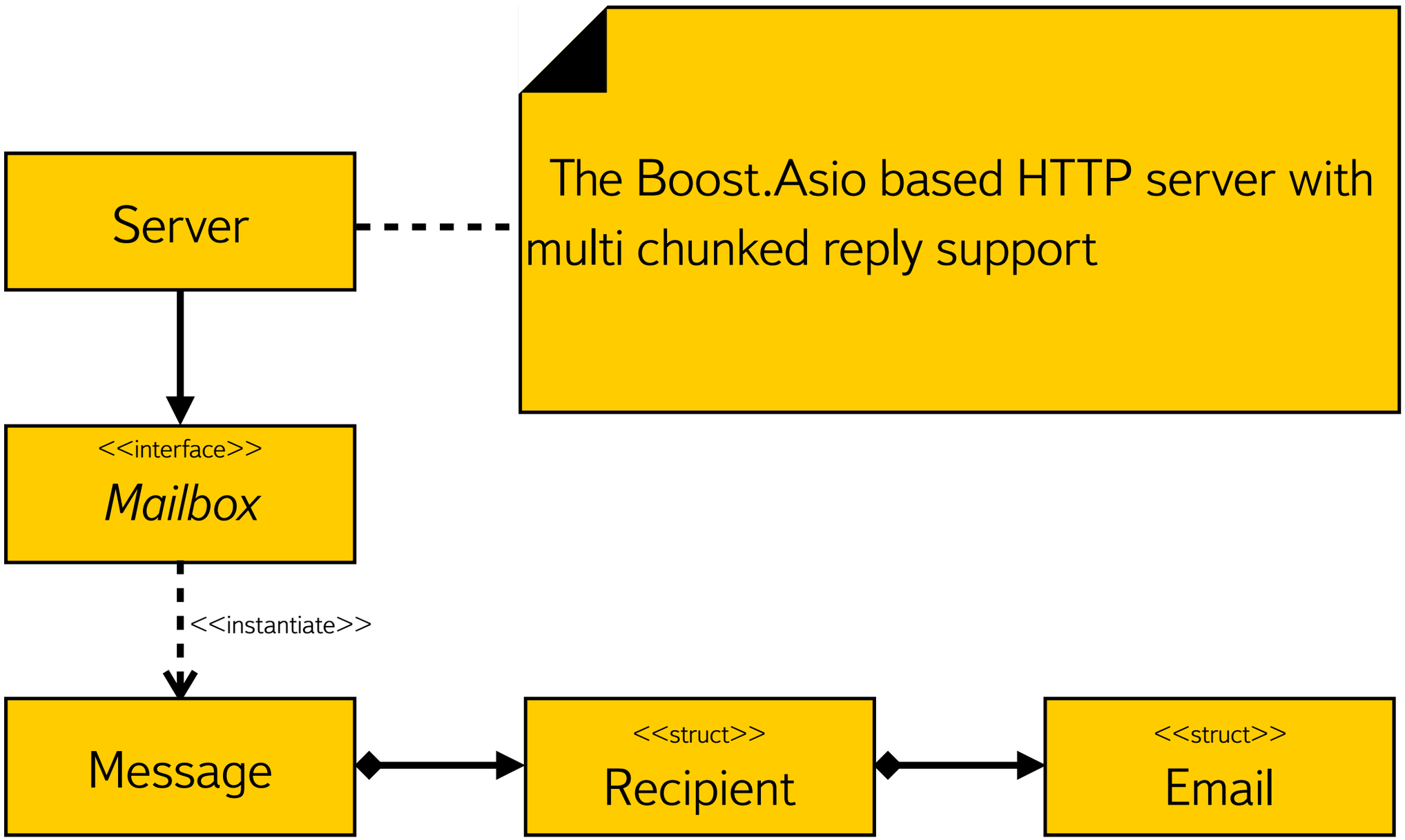
The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service Model

Mailbox access via web service model

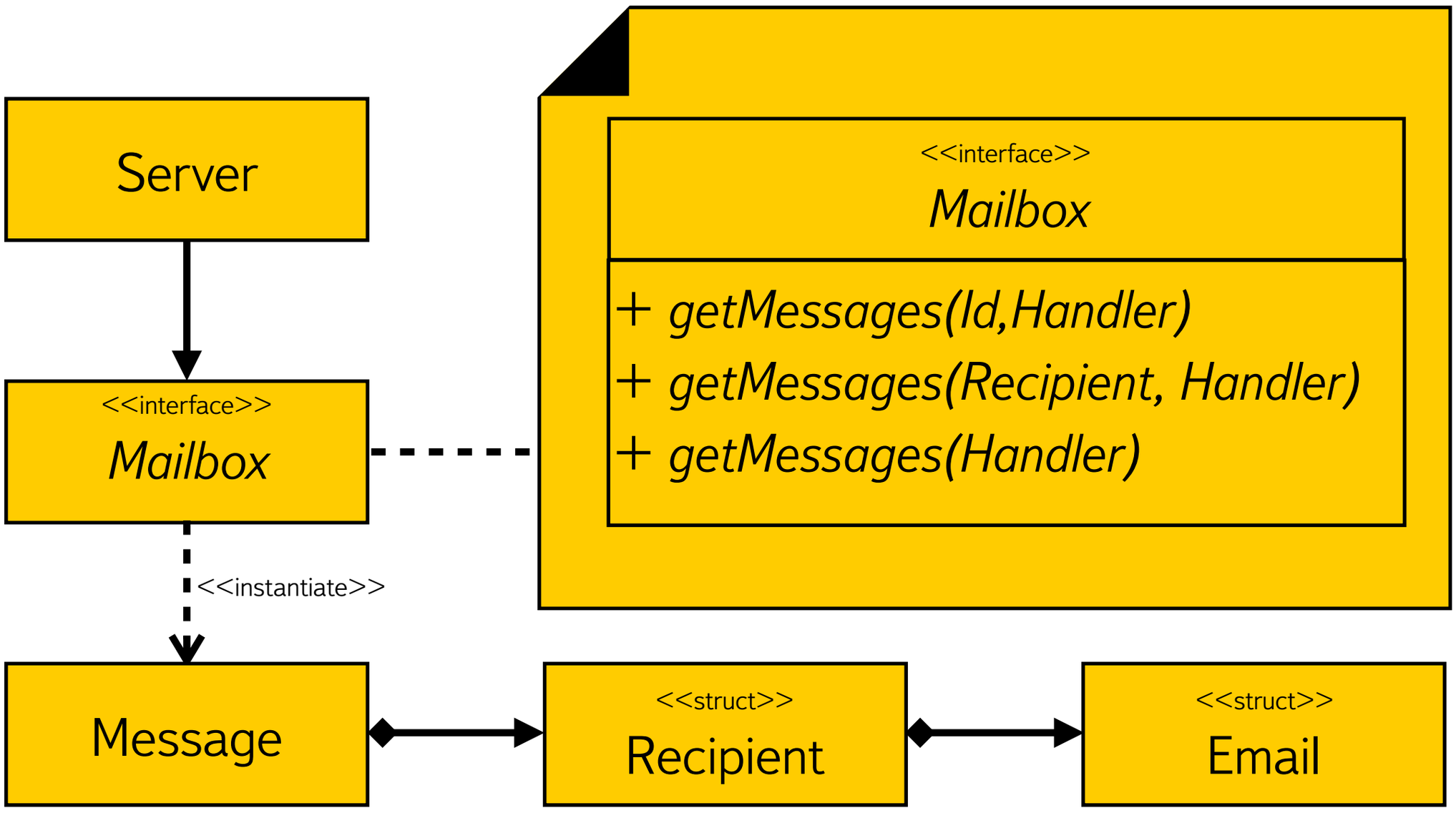
We examine the web service which provides simple API to access user's mailbox

- *messages/* - return all the mailbox messages
- *messages/id* - returns a message by id
- *messages/recipient* - returns messages by recipient - **used in benchmark**

The Service Architecture



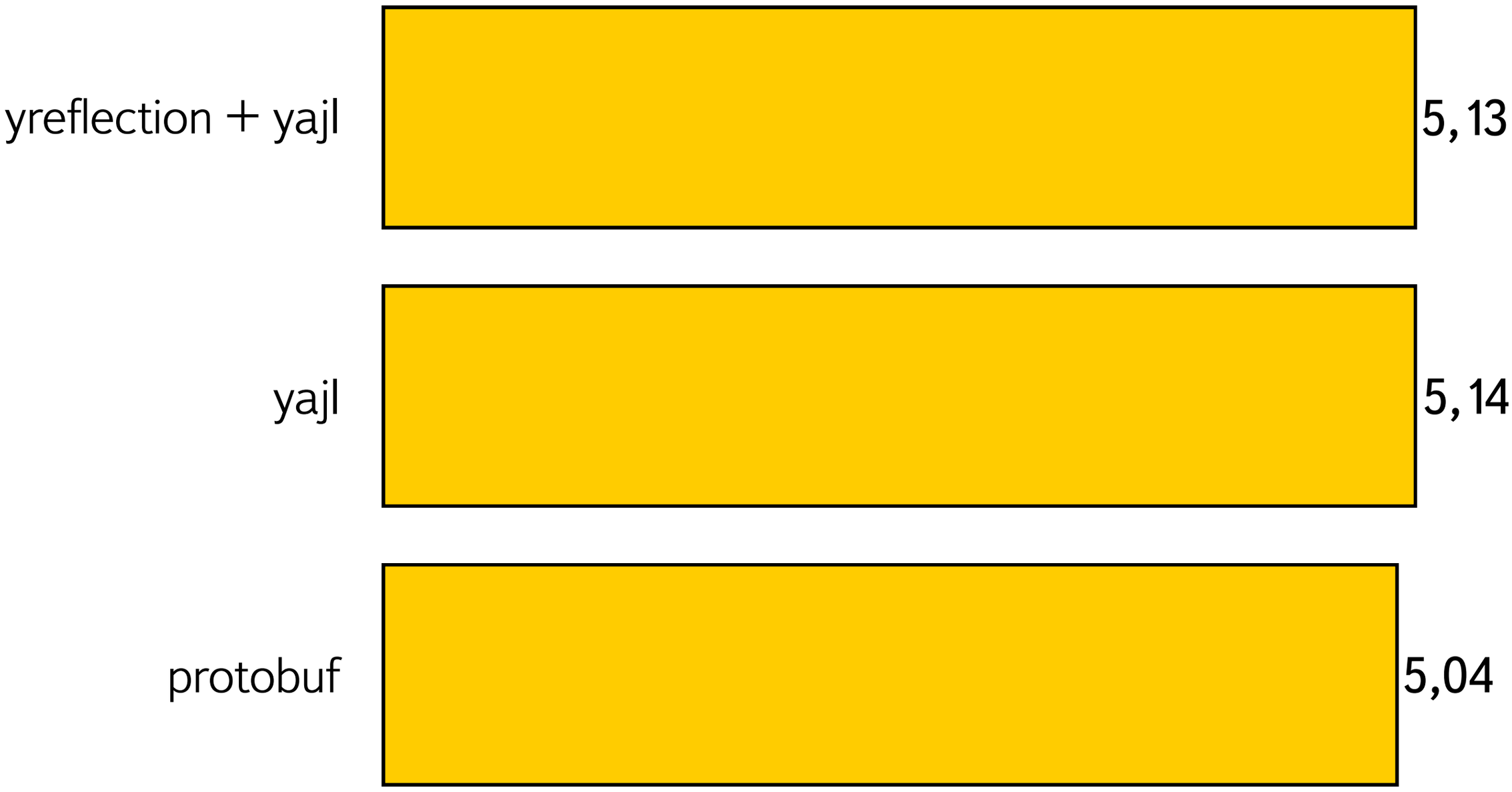
The Service Architecture



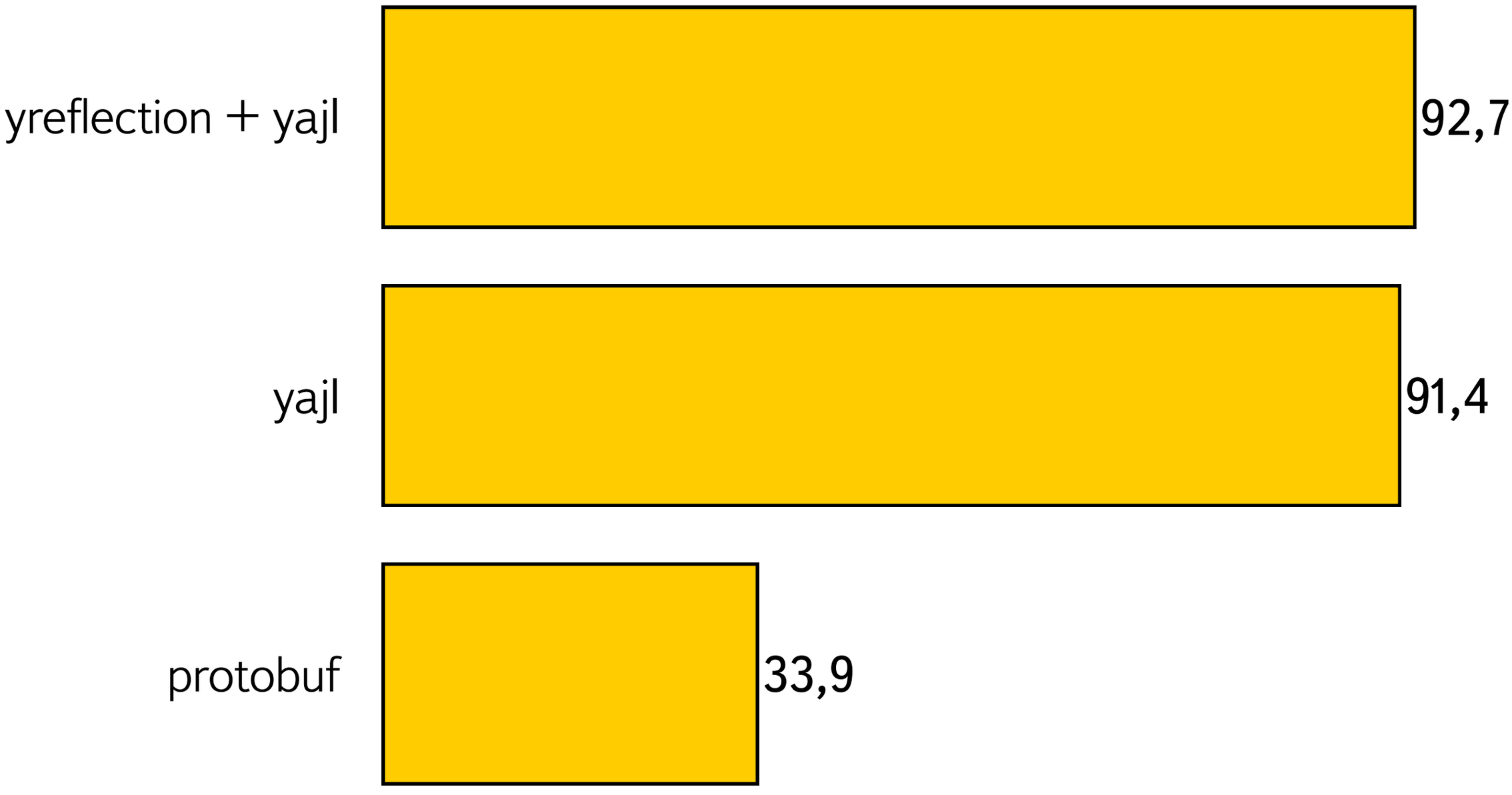
The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

Results

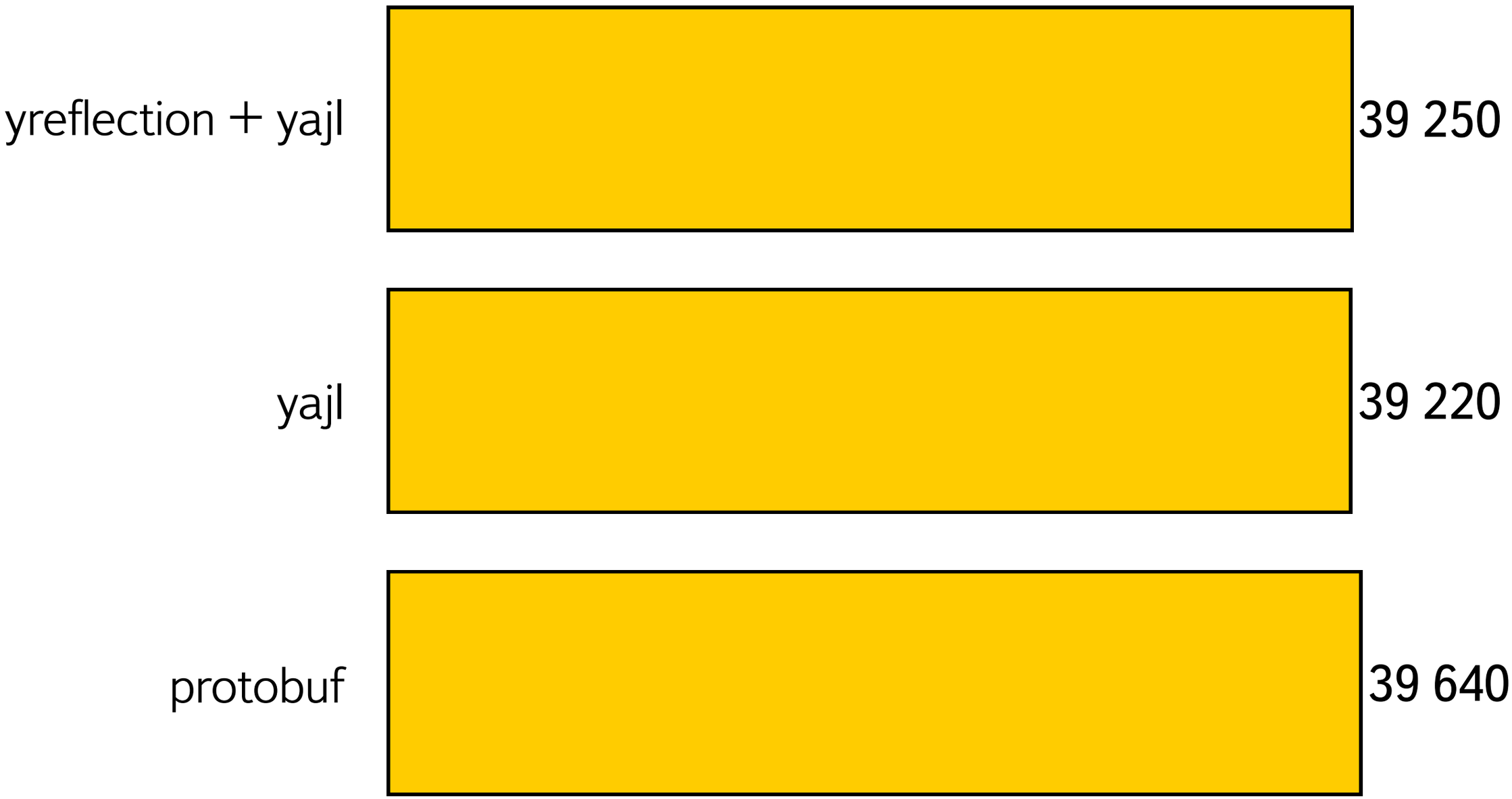
Latency, ms



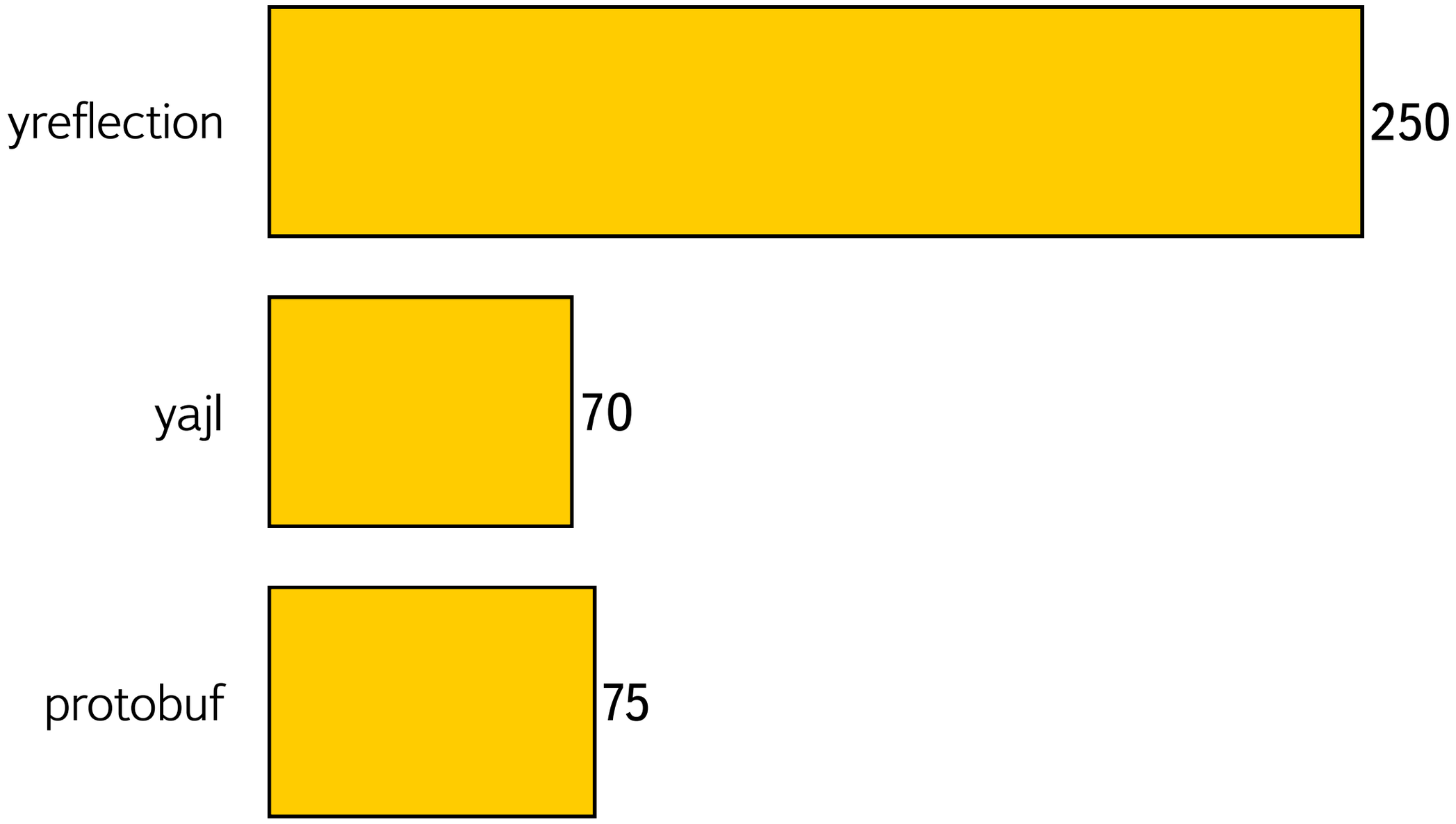
CPU consumption



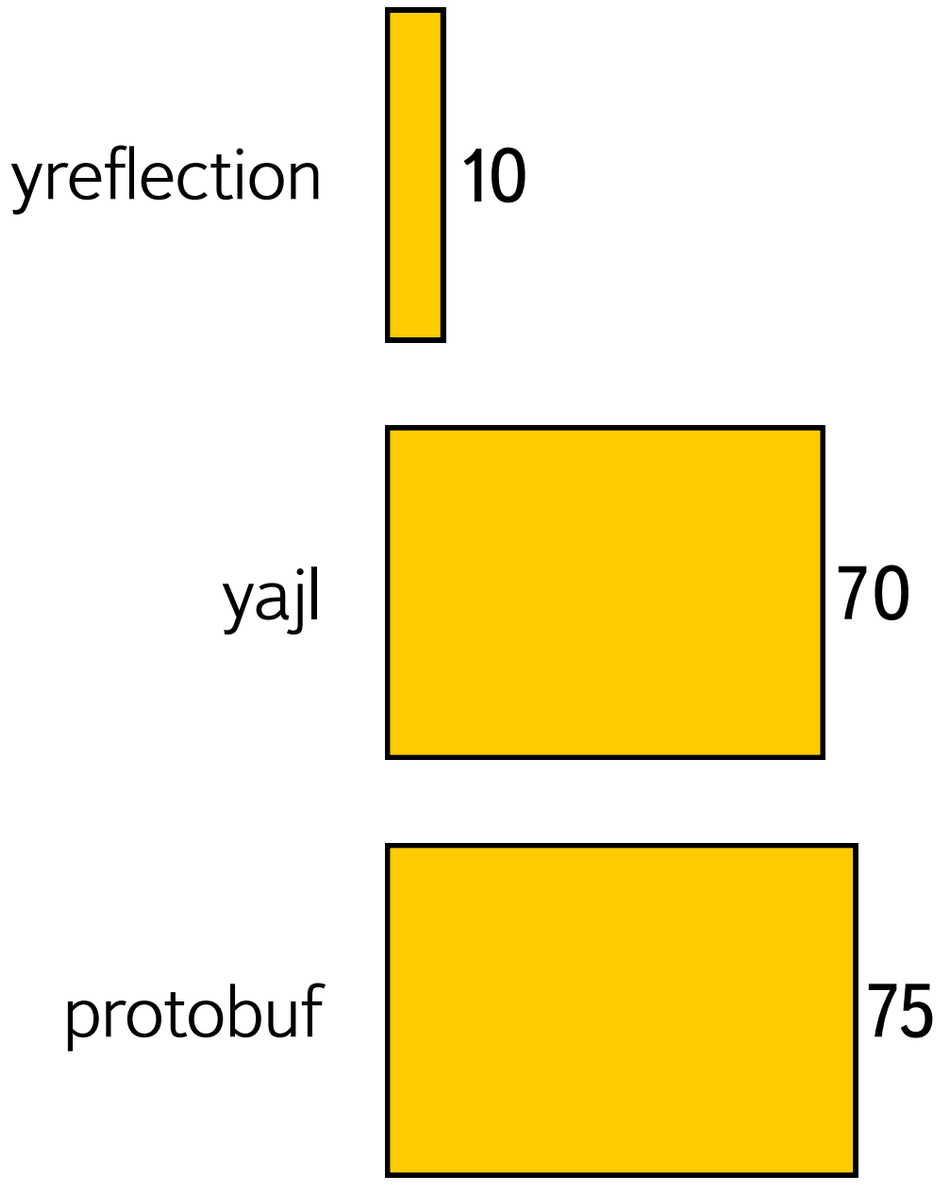
Memory, KB



LOC



LOC



The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

💔 The Boost.Fusion Issues

Defining different versions of response

> Let’s try to use BOOST_FUSION_ADAPT_NAMED:

```
struct Struct {
    Array array;
    int id;
};
struct Response {
    Struct b;
};

BOOST_FUSION_ADAPT_STRUCT(Struct, id, array)
BOOST_FUSION_ADAPT_STRUCT(Response, b)
//Old version of response
BOOST_FUSION_ADAPT_STRUCT_NAMED(Struct, StructV1, id)
BOOST_FUSION_ADAPT_STRUCT_NAMED(Response, ResponseV1, b)
```

Something goes wrong

> Oops, we got the warning:

```
reflection/struct.cc:105:1: warning: returning
reference to local temporary object
[-Wreturn-stack-address]
```

access::struct_member::apply() problem

> Simplified macro expansion of BOOST_FUSION_ADAPT_NAMED

```
struct access::struct_member< ResponseV1 , 0 > {
    typedef StructV1 attribute_type;
    template<typename Seq>
    struct apply {
        typedef typename const_if<attribute_type, Seq> inner_type;
        typedef typename add_reference<inner_type>::type type;
                                        //yes, it adds reference always


        constexpr static type call(Seq& seq) {
            return seq.obj.b; //for a view we return a ref to the local
        }
    };
};
```

The solution is

> We could return by-field when «Seq» type is a view, like this

```
struct access::struct_member< ResponseV1 , 0 > {
    typedef StructV1 attribute_type;
    template<typename Seq>
    struct apply {
        typedef typename const_if<attribute_type, Seq> inner_type;
        typedef typename mpl::eval_if< struct_is_view<attribute_type>,
                add_reference<inner_type>,
                inner_type
            >::type
        type;
        constexpr static type call(Seq& seq) {
            return seq.obj.b; //for a view we return by field
        }
    };
};
```

View problem

> Generated view cannot be used with const objects

```
BOOST_FUSION_ADAPT_STRUCT_NAMED( Struct, StructV1, id )

// ...

namespace  {
    struct StructV1 {
        constexpr StructV1(Struct& in_obj) // No const support
         : obj(in_obj) {}
        Struct& obj;
    private:
        StructV1& operator= (StructV1 const&);
    };
}
```

Probable solution to View problem

> We can define additional view for const objects

```
BOOST_FUSION_ADAPT_STRUCT_NAMED( Struct, StructV1, id )
BOOST_FUSION_ADAPT_STRUCT_NAMED( const Struct, StructCV1, id )
// ...

namespace  {
    struct StructV1 {
        constexpr StructV1(Struct& in_obj);
    };

    struct StructCV1 {
        constexpr StructCV1(const Struct& in_obj);
    };
}
```

> Doesn’t scale well when adapting nested structures

boost_fusion_adapt_adt_impl_set()

> Boost.Fusion doesn't support moving field into setter

```
class A {
    void set(std::string val);
};

BOOST_FUSION_ADAPT_ADT(A, (obj.get(), obj.set(forward<Val>(val))))
```

> Because part of macro expansion for setter looks like this:

```
template<class Val>
static void boost_fusion_adapt_adt_impl_set(A& obj, Val const& val) {
    obj.set(forward<Val>(val)); // No perfect forwarding, sorry
}
```

Solution is

> Boost.Fusion doesn’t support moving field into setter

```
class A {
    void set(std::string val);
};

BOOST_FUSION_ADAPT_ADT(A, (obj.get(), obj.set(forward<Val>(val))))
```

> Because part of macro expansion for setter looks like this:

```
template<class Val>
static void boost_fusion_adapt_adt_impl_set(A& obj, Val &&      val) {
    obj.set(forward<Val>(val)); // Perfect forwarding works here
}
```

BOOST_FUSION_ADAPT_ADT - No name

> You can't link a name to getter/setter pair

```
struct A {
    void field(std::string const& val);
    std::string const& field() const;
};

BOOST_FUSION_ADAPT_ADT(A, (obj.field(), obj.field(val)))
auto json = JsonWriter.apply(A("foo"), "A");
```

> What we want:

```
{
  "A": {
     "field": "foo"
  }
}
```

> What we get:

```
{
  "A": [
     "foo"
  ]
}
```

First non-optimal solution

> We emulate named field by proxying it with std::pair of name and value

```
inline std::string stripMethodName(std::string name);

#define YR_CALL_WITH_SPECIFIC_NAME(fun, name) \
    std::make_pair(name, obj.fun())

#define YR_CALL_WITH_NAME(fun) \
    YR_CALL_WITH_SPECIFIC_NAME(fun, stripMethodName(#fun))

#define YR_CALL_SET_WITH_NAME(fun) obj.fun( val.second );

BOOST_FUSION_ADAPT_ADT(A,
    (YR_CALL_SET_WITH_NAME(field), YR_CALL_WITH_NAME(field)) )
```

> Copy of name (and function result, if given by ref) made; fragile

A near-optimal solution is

> The solution by Sehe can be found on Stack Overflow [1]

```
#define MY_ADT_MEMBER_NAME(CLASSNAME, IDX, MEMBERNAME)\
    namespace boost { namespace fusion { namespace extension {\
    template <> struct struct_member_name<CLASSNAME, IDX> {\
        typedef char const* type;\
        static type call() { return #MEMBERNAME; }\
    }; } } }

BOOST_FUSION_ADAPT_ADT(A, obj.field(), obj.field(val))
MY_ADT_MEMBER_NAME(A, 0, field)
```

> Have to declare it separately from main macro and point field's index in MPL sequence manually

What we want

> Automatic deduction from getter's name

```
struct A {
    void field(std::string const& val);
    std::string const& field() const;
    std::string const& readOnlyField() const;
};
```

```
YREFLECTION_ADAPT_ADT_PROPERTIES(A,
    field, readOnlyField ) // Name can be deduced from getter name
```

> JSON:

```
{
   "field": "foo"
}
```

What we want

> Explicitly set field name

```
struct A {
    void field(std::string const& val);
    std::string const& field() const;
    std::string const& readOnlyField() const;
};
```

```
YREFLECTION_ADAPT_ADT_SET_GET(A,
    (field, getField, setField)
    (readOnlyField, getReadOnlyField) ) // Name is specified directly
```

> JSON:

```
{
   "field": "foo",
   "readOnlyField": "bar",
}
```

What we want

> External set and get functions

```
struct A {
    void field(std::string const& val);
    std::string const& field() const;
    std::string const& readOnlyField() const;
};
```

```
Item str2item(std::string); // Converts string to the brand new Item
std::string item2str(Item); // Converts Item to the old string

YREFLECTION_ADAPT_ADT_SET_GET(A,
    (field, obj.setField(item2str(val)), str2item(obj.getField())))
```

The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

Summary

Conclusions

Yreflection:

- Has near-zero runtime overhead
- Allows to write generic representation code
- Is compact to use for clients

Summary

We love Boost.Fusion

- It gives us convenient mechanism for introspection
- But it needs to be a little bit fixed and changed

We want native introspection in C++

- Boost.Fusion adaptation is good, but requires heavy macros
- We don’t want provide compile-time information to compiler

What we do next

- Features to be implemented
  - Documentation
  - ADT with names
  - Magic for SAX parsers

The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

🐼 Questions?

Links

1. Stack Overflow post: http://stackoverflow.com/questions/26380420/boost-fusion-sequence-type-and-name-identification-for-structs-and-class
2. Boost.Fusion documentation: http://www.boost.org/doc/libs/1_60_0/libs/fusion/doc/html/index.html
3. Our dedicated project on Github: https://github.com/thed636/reflection

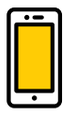
The Effective Structured Data Marshalling/Demarshalling Through Boost.Fusion Introspection In A High Performance Web Service

Спасибо (Thank you)!

Contacts

Sergei Khandrikov

Senior Software Developer



@S_Khandrikov

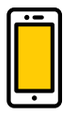
thed636@yandex-team.ru

thed636

Contacts

Piotr Reznikov

Software Developer



@

prez@yandex-team.ru

pyotr.reznikov