Using Functional Programming patterns to build a clean and simple HTTP routing API

MUREX™

How would you feel?

```
Response process(const Request& rq) {
  const auto& uri = rq.uri();
  std::regex foo_re{"/v1/foos(/([0-9]+))?"};
  std::smatch foo_match;
  if (std::regex_match(uri, foo_match, foo_re)) {
     std::ssub_match id = foo_match[2];
     if (id.length() == 0) {
        if (rq.method() == "GET")
           return {list_all_foos(), 200};
        if (rq.method() == "POST") {
           foo new_foo =
              nlohmann::json::parse(rq.body());
           all_foos.push_back(new_foo);
           return {new_foo, 201};
        }
        return {nullptr, 405};
     }
    if (rq.method() != "GET")
        return {nullptr, 405};
     auto foo = find_foo(std::stoi(id.str()));
     if (foo)
        return {*foo, 200};
     return {nullptr, 404};
  }
  if (rq.uri() == "/v1/bars") {
     if (rq.method() != "GET")
        return {nullptr, 405};
     return {list_all_bars(), 200};
  }
  return {"", 404};
}
```

Better now?

```
auto version = to_path("v1");
auto foos = version / "foos";
auto api = router(
  GET (foos               , list_all_foos),
  GET (foos / param<int>(), get_foo_by_id),
  POST(foos / body<foo>() , insert_new_foo),
  GET (version / "bars"   , list_all_bars)
);
```

About us

@jeremydemeule

- 9 years at Murex
- CDBC poster

@quduval

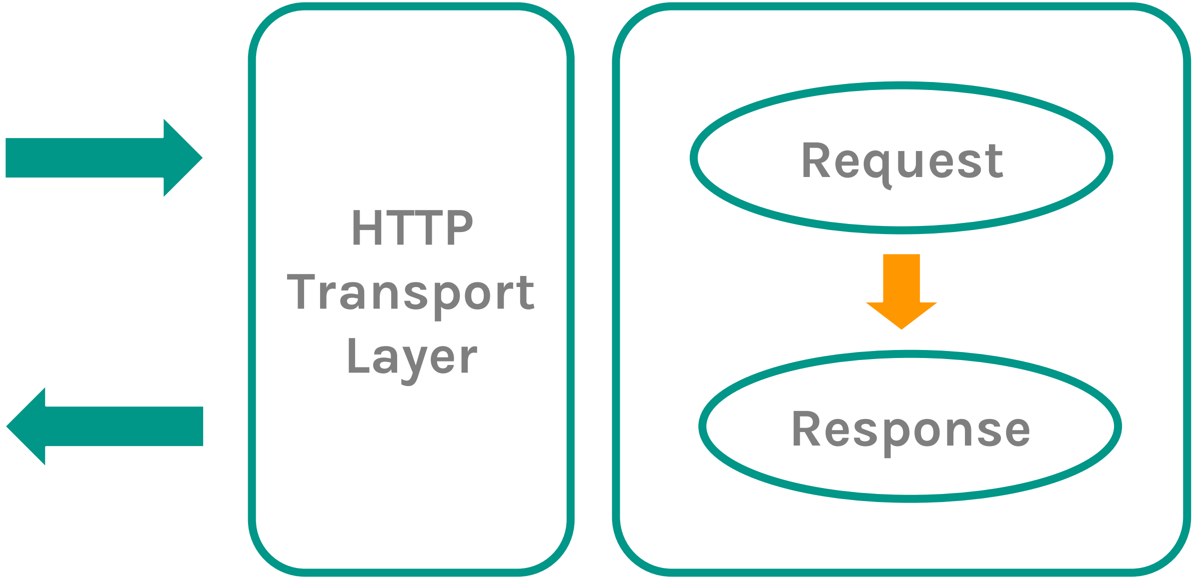
- 6 years at Murex
- deque.blog

1.

Describing the problem

We need a HTTP routing library

A simple HTTP Server



A simple HTTP server

```
# GET /v1/foos

[{"id":1},{"id":2},{"id":3}]
```

A simple HTTP server

```
Response process(const Request& rq) {
  if (rq.uri() == "/v1/foos") {
     return {list_all_foos(), 200};
  }
  return {"", 404};
}
```

A simple HTTP server

```
Response process(const Request& rq) {
  if (rq.uri() == "/v1/foos") {
    return {list_all_foos(), 200};
  }
  return {"", 404};
}
```

A simple HTTP server

```
Response process(const Request& rq) {
  if (rq.uri() == "/v1/foos") {
     return {list_all_foos(), 200};
  }
  return {"", 404};
}
```

A simple HTTP server

```
Response process(const Request& rq) {
  if (rq.uri() == "/v1/foos") {
     return {list_all_foos(), 200};
  }
  return {"", 404};
}
```

Spawn more routes

```
Response process(const Request& rq) {
  if (rq.uri() == "/v1/foos") {
     return {list_all_foos(), 200};
  }
  if (rq.uri() == "/v1/bars") {
     return {list_all_bars(), 200};
  }
  return {"", 404};
}
```

Capturing Parameters

```
# GET /v1/foos/2

{"id":2}
```

Capturing Parameters

```
std::regex foo_re{"/v1/foos(/([0-9]+))?"};
if (std::smatch match;
    std::regex_match(rq.uri(), match, foo_re))
{
  std::ssub_match id = match[2];
  if (id.length() == 0) return {list_all_foos(), 200};
  auto foo = find_foo(std::stoi(id.str()));
  return foo ? {*foo, 200} : {nullptr, 404};
}
```

Capturing Parameters

```
std::regex foo_re{"/v1/foos(/([0-9]+))?"};
if (std::smatch match;
    std::regex_match(rq.uri(), match, foo_re))
{
  std::ssub_match id = match[2];
  if (id.length() == 0) return {list_all_foos(), 200};
  auto foo = find_foo(std::stoi(id.str()));
  return foo ? {*foo, 200} : {nullptr, 404};
}
```

Adding HTTP Verbs

```
# POST /v1/foos -d '{"id":42}'

# GET /v1/foos
[{"id":1},{"id":2},{"id":3},{"id":42}]
```

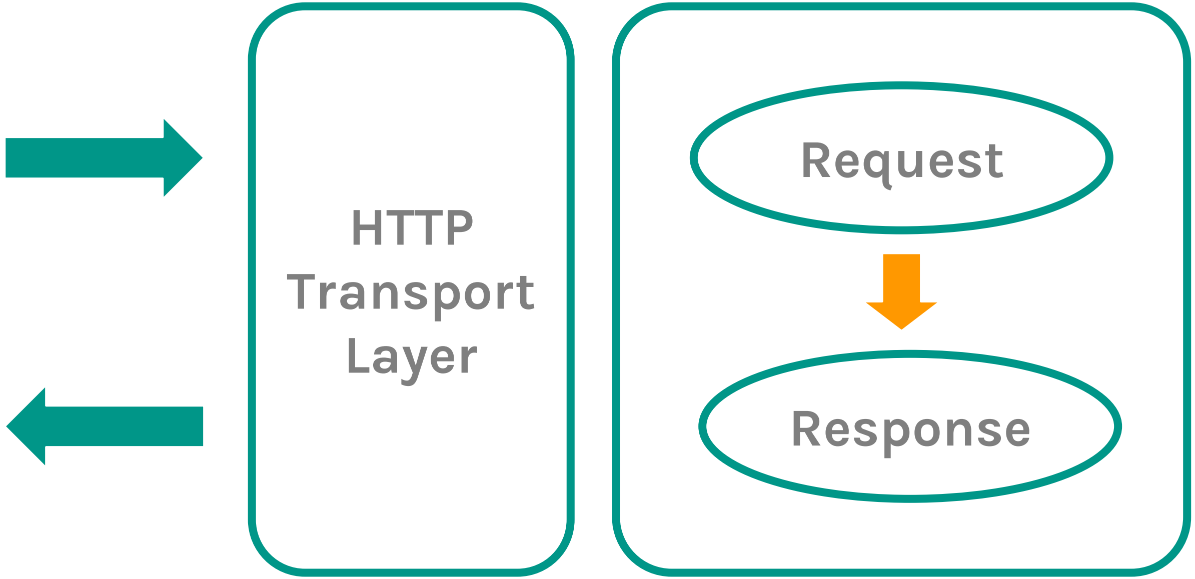
Adding HTTP Verbs

```
Response process(const Request& rq) {
  const auto& uri = rq.uri();
  std::regex foo_re{"/v1/foos(/([0-9]+))?"};
  std::smatch foo_match;
  if (std::regex_match(uri, foo_match, foo_re)) {
    std::ssub_match id = foo_match[2];
    if (id.length() == 0) {
      if (rq.method() == "GET")
        return {list_all_foos(), 200};
      if (rq.method() == "POST") {
        foo new_foo =
          nlohmann::json::parse(rq.body());
        all_foos.push_back(new_foo);
        return {new_foo, 201};
      }
      return {nullptr, 405};
    }
    if (rq.method() != "GET")
      return {nullptr, 405};
    auto foo = find_foo(std::stoi(id.str()));
    if (foo)
      return {*foo, 200};
    return {nullptr, 404};
  }
  if (rq.uri() == "/v1/bars") {
    if (rq.method() != "GET")
      return {nullptr, 405};
    return {list_all_bars(), 200};
  }
  return {"", 404};
}
```

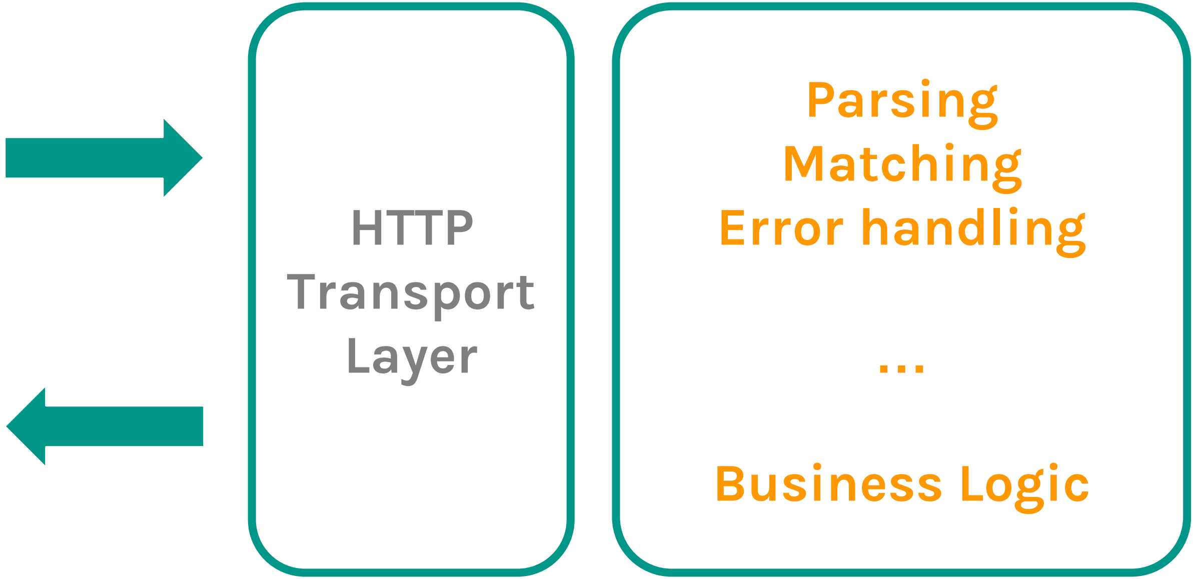
Summary

- **4** routes
- **30** lines of spaghetti code
- **8** conditional branching
- **3** nested levels of indentation

From a good state...



… right into a mess



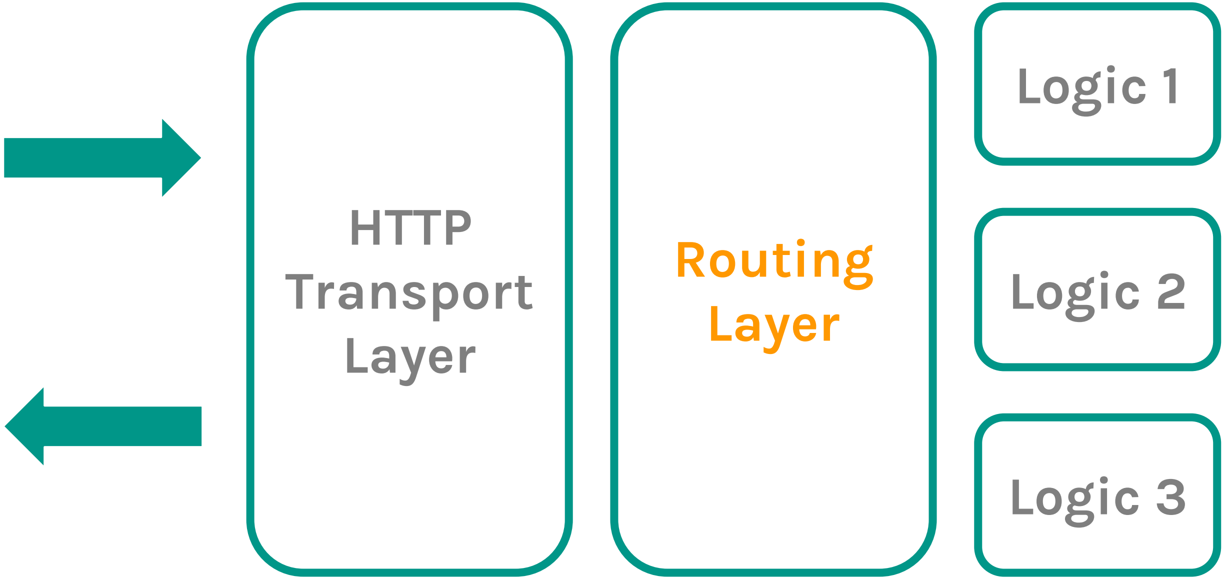
HTTP Transport Layer

Parsing
Matching
Error handling

…

Business Logic

The missing abstraction



2.

A first approach

Annotation
& Reflection
based

Introducing AOP

```
Response process(const Request& rq) {
  if (rq.uri() == "/v1/foos") {
     return {list_all_foos(), 200};
  }
  return {"", 404};
}
```

Introducing AOP

# GET /v1/foos

```
[[RequestMapping("/v1/foos", GET)]]
Response foos() {
  return list_all_foos();
}
```

Capturing parameters

# GET /v1/foos/2

```
[[RequestMapping("/v1/foos/{id}", GET)]]
Response foo_by_id([[PathVariable]] int id) {
  return find_foo(id);
}
```

Plugging to the framework

```
[[RequestController]]
struct FooController {

  [[RequestMapping("/v1/foos", GET)]]
  Response foos();

  [[RequestMapping("/v1/foos/{id}", GET)]]
  Response foo_by_id([[PathVariable]] int id);
};
```

Plugging to the framework

```
controller FooController {

   [[RequestMapping("/v1/foos", GET)]]
   Response foos();

   [[RequestMapping("/v1/foos/{id}", GET)]]
   Response foo_by_id([[PathVariable]] int id);
};
```

Declaring dependencies

```
controller FooController {
   [[Inject]] FooService* m_foos;

   [[RequestMapping("/v1/foos", GET)]]
   Response foos() {
     return m_foos->list_all_foos();
   }
};
```

Scaling the pattern

```
controller FooController {
   [[Inject]] FooService* m_foos;
};

controller BarController {
   [[Inject]] BarService* m_bars;
};
```

Not idiomatic for C++

- **Annotation**: runtime reflection
- **Interface**: runtime polymorphism
- **Dependency injection**: runtime

Improving on AOP

- Limited composition (e.g. URI)
- Limited cohesion
  - URI scattered between controller
- High coupling via annotations

> “
>
> You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Joe Armstrong
(Creator of Erlang)

> “
>
> You wanted a route but what you got was a controller holding the route and the entire framework.
>
> Jeremy & Quentin

Keep the abstraction, but...

- Idiomatic C++ implementation
- Increase transparency & composition
- Reduce coupling to technology

3.

Functional Design

Composition over annotation

Expose meaningful concepts

1

Clear Semantic

○ Abstract definition

○ Precise meaning

○ 1 concept = 1 function

Avoid banana-gorilla syndrome

2

Compose

○ 1 concept ⇒ 1 entity

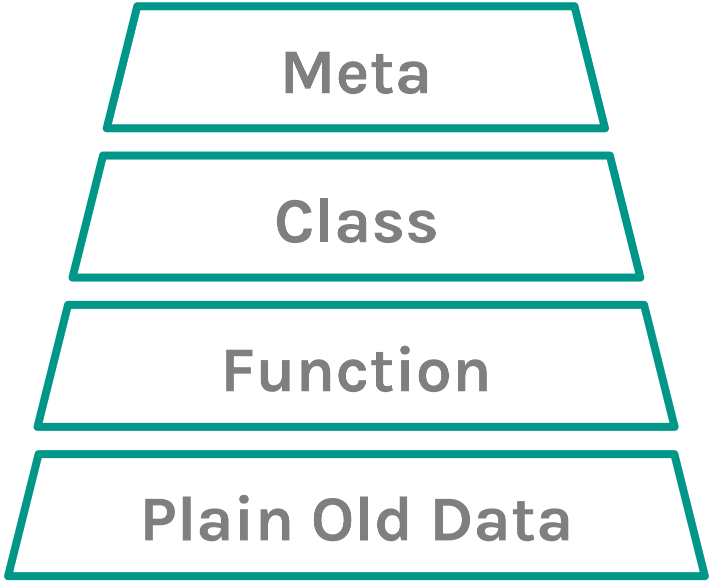
○ Simple building blocks

○ Build complex behaviors

Linking abstractions to real code

3

Simple Constructs



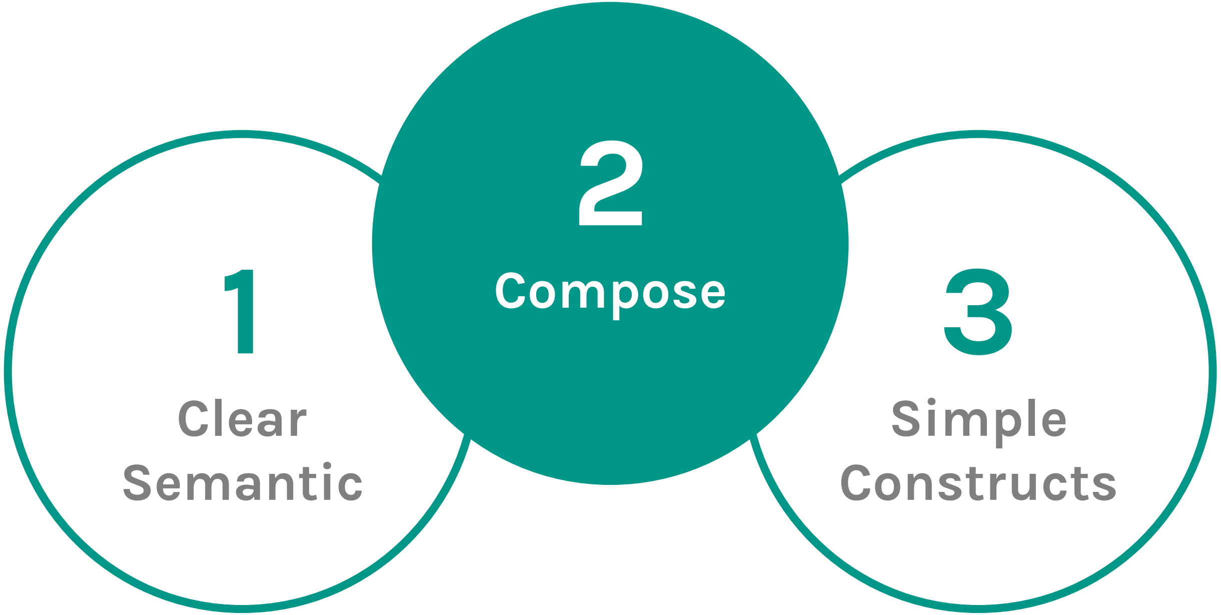
Meta

Class

Function

Plain Old Data

Overall approach

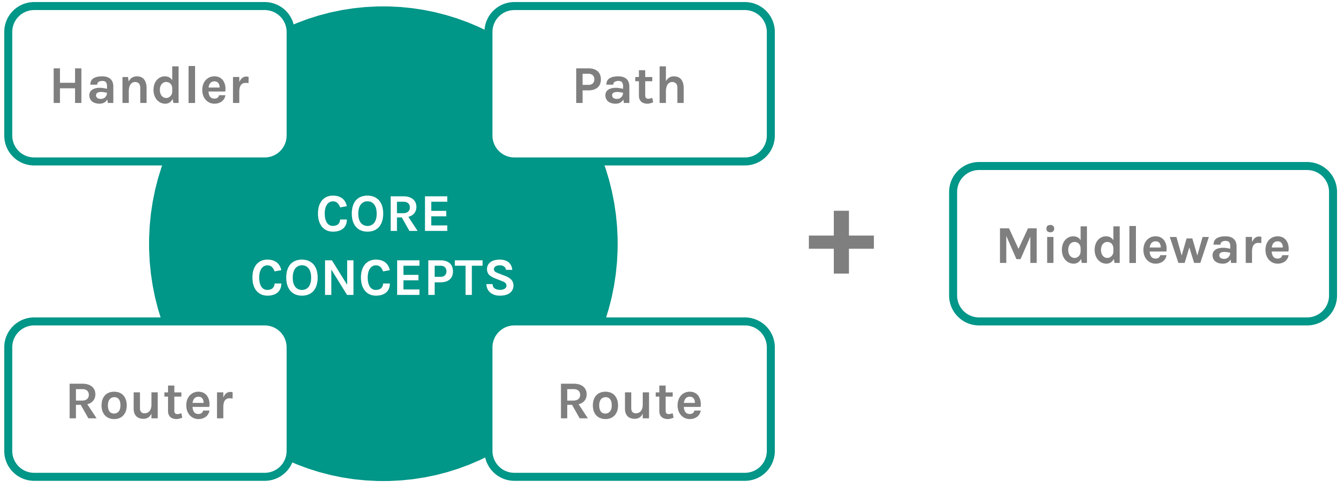


1 Clear Semantic

2 Compose

3 Simple Constructs

Concepts of HTTP routing



Handler

Path

CORE CONCEPTS

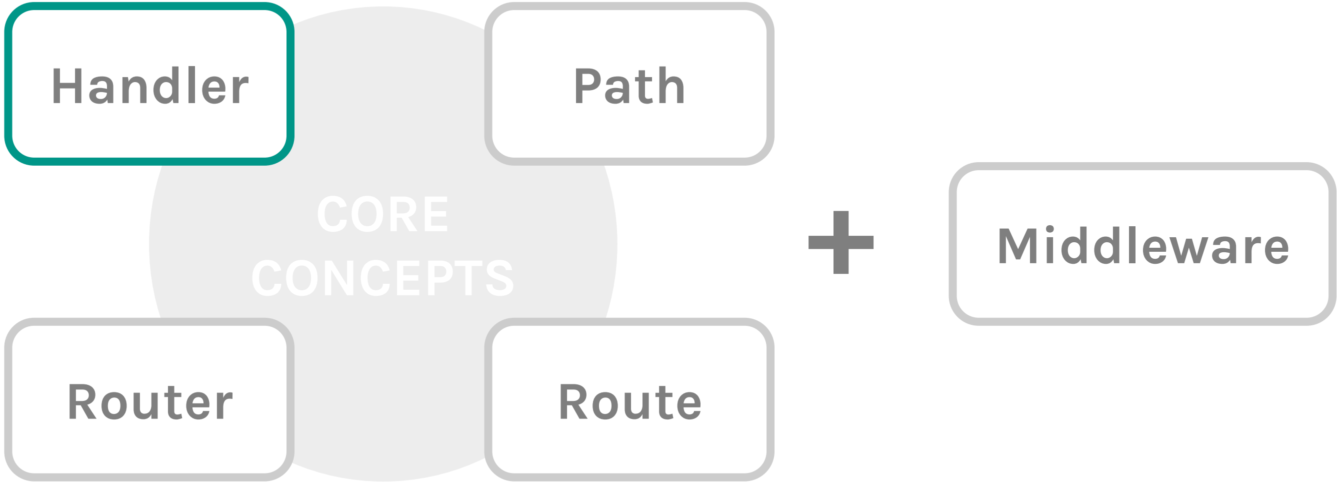
Router

Route

+

Middleware

Concepts of HTTP routing



Handler

Path

CORE CONCEPTS

Router

Route

+

Middleware

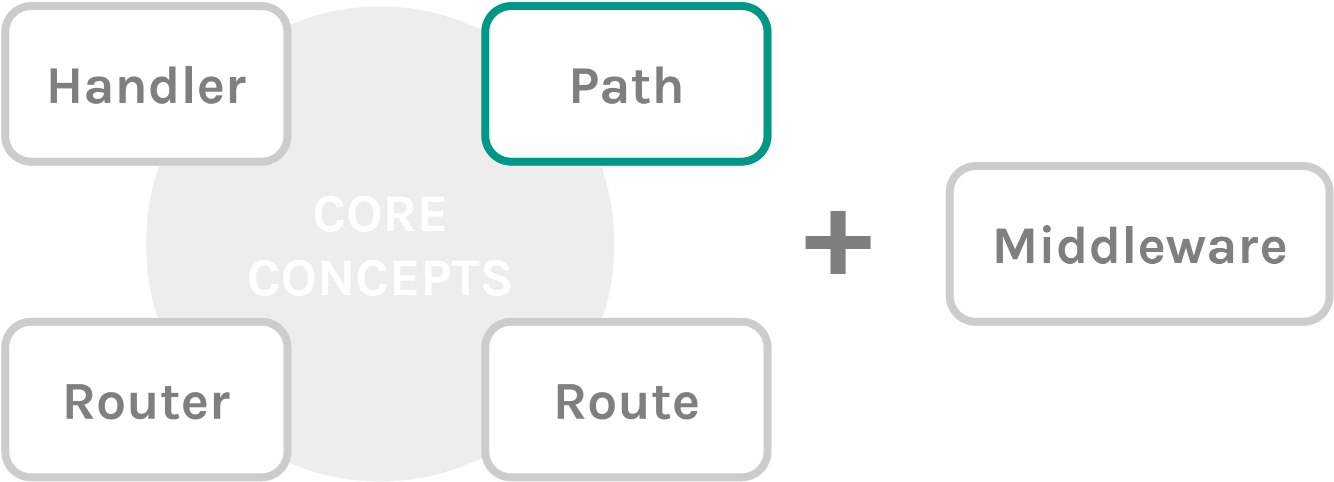
Handler = Unit of Business Logic

- Answer a specific request
- Take URI path parameters
- Return appropriate answer

Request -> Params -> Response

```
auto get_foo_by_id =
  [](const Request& rq, int id) -> Response
{
    auto foo = find_foo(id);
    if (foo) return {*foo, OK};
    return {nullptr, NotFound};
}
```

Concepts of HTTP routing



Path = {Set of accepted URI}

- Like "/v1/foos/([0-9]+)"

  Path = URI -> Bool

Path = {Set of accepted URI}

○ Extract parameters & types

Path = URI -> Params?

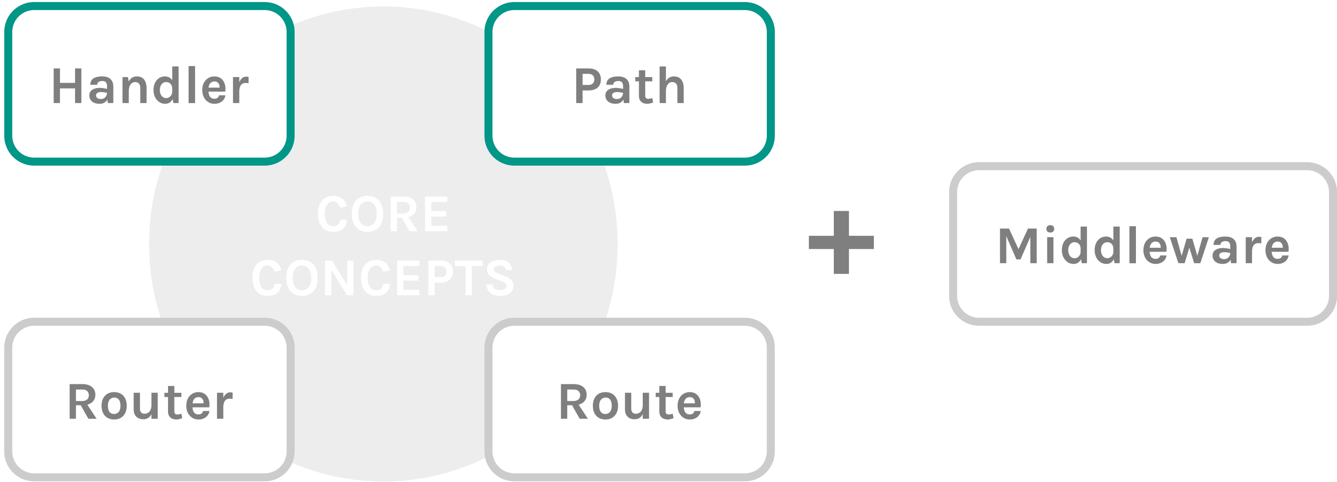
Path as data

```
auto all = to_path("v1") / "foos";

auto by_id = to_path("v1") / "foos" / param<int>();
```

Path + Path = Path

```
auto all = to_path("v1") / "foos";

auto by_id = all / param<int>();

auto by_name = all / param<std::string>();
```

Concepts of HTTP routing



Path

+

Handler

URI -> Params?

+

Request -> Params -> Response

Request -> Params?

+

Request -> Params -> Response

Request ->

Params?

+

Params -> Response

Request -> (

Params?

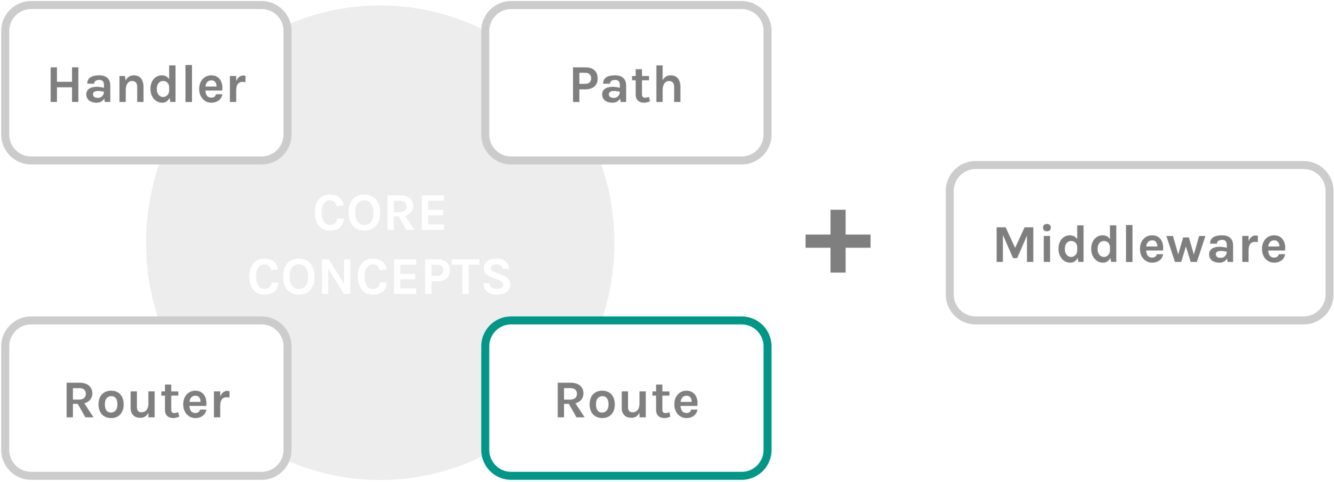
+

Params? -> Response?

)

Request -> Response?

Concepts of HTTP routing



Handler

Path

CORE CONCEPTS

Router

Route

+

Middleware

Route = Request -> Response?

- Match HTTP Verb
- Match URI against Path
- Call the Handler

Route = Request -> Response?

```
GET(version / "foos" / param<int>(),
    [](const Request& rq, int id)
    {
      //Implementation
    });
```

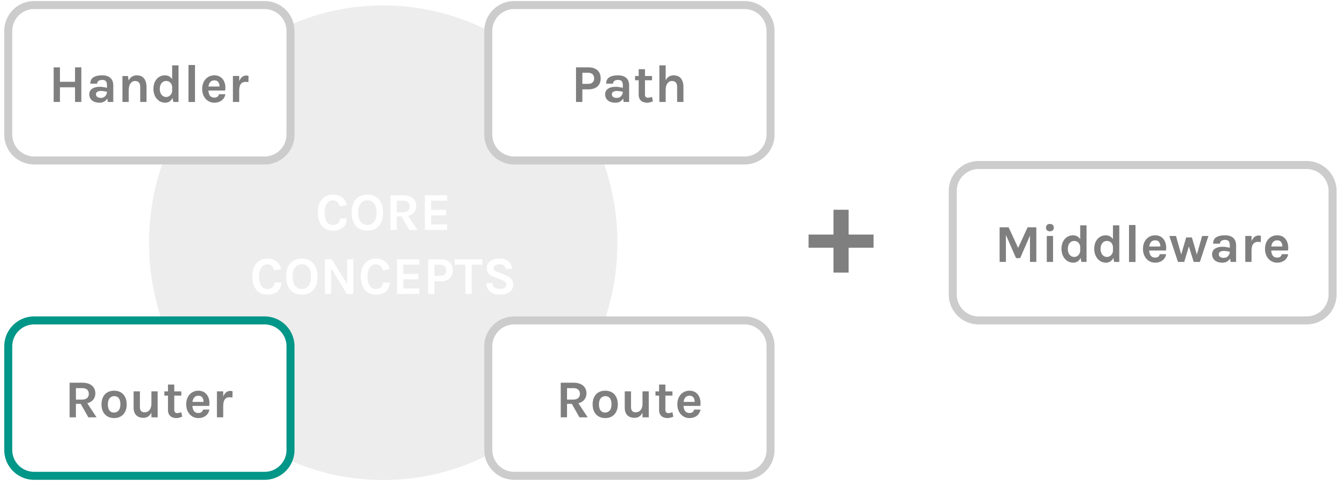
Route = Request -> Response?

```
GET(version / "foos" / param<int>(),
    [](const Request& rq, int id)
    {
      //Implementation
    });
```

Route = Request -> Response?

```
GET(version / "foos" / param<int>(),
    [](const Request& rq, int id)
    {
      //Implementation
    });
```

Concepts of HTTP routing



Handler

Path

CORE CONCEPTS

Router

Route

+

Middleware

Router

- Holds several routes
- Has to return an answer

Router = Request -> Response

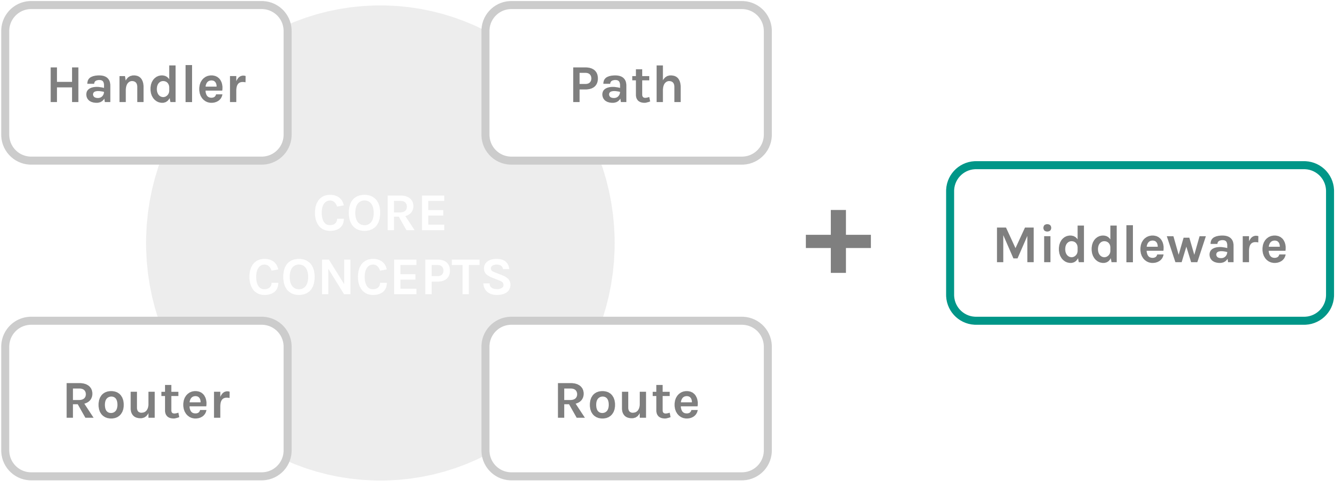
Router = Request -> Response

```
auto version = to_path("v1");
auto foos = version / "foos";
auto api = router(
  GET (foos               , list_all_foos),
  GET (foos / param<int>(), get_foo_by_id),
  POST(foos / body<foo>() , insert_new_foo),
  GET (version / "bars"   , list_all_bars)
);
```

Router = Request -> Response

```
auto version = to_path("v1");
auto foos = version / "foos";
auto api = router(
  GET (foos               , list_all_foos),
  GET (foos / param<int>(), get_foo_by_id),
  POST(foos / body<foo>() , insert_new_foo),
  GET (version / "bars"   , list_all_bars)
);
```

Concepts of HTTP routing



Cross cutting concerns

```
GET (v1 / "foos", list_all_foos)

GET (v1 / "bars", list_all_bars)
```

Middleware = Handler -> Handler

```
GET (v1 / "foos", with_logs(list_all_foos))

GET (v1 / "bars", with_rights(list_all_bars))
```

Middleware = Handler -> Handler

```cpp
auto with_logs = [&logger] (auto&& handler) {
  return [&](Request const& req, auto&& ...args) {
    logger << req.uri();
    return handler(req, args...);
  };
};
```

Composing middleware

```
auto standard_middleware
  = with_rights
  | with_logs;


GET (bars, standard_middleware(list_all_bars))
```

4.

Result & Benefits

Decoupling
Cohesion
Testability

Full HTTP router

```
auto get_api() {
  auto version = to_path("v1");
  auto foos = version / "foos";
  auto api = router(
    GET (foos               , get_list_all_foos),
    GET (foos / param<int>(), get_foo_by_id),
    POST(foos / body<foo>() , post_new_foo),
    GET (version / "bars"   , get_list_all_bars)
  );
}
```

```
Response get_list_all_foos(const Request& rq) {
   return {list_all_foos(), OK};
}

Response get_foo_by_id(const Request& rq, int id) {
   auto foo = find_foo(id);
   if (foo) return {*foo, OK};
   return {nullptr, NotFound};
}

Response post_new_foo(const Request& rq,
                      const foo& foo) {
   all_foos.push_back(foo);
   return {foo, Created};
}

Response get_list_all_bars(const Request& rq) {
   return {list_all_bars(), OK};
}
```

Summary

- **4** routes
- **29** lines of code
- **1** conditional branching
- **1** nested level of indentation

Decoupling & Cohesion

```
auto get_api() {
  auto version = to_path("v1");
  auto foos = version / "foos";
  return router(
    GET (foos                  , get_list_all_foos),
    GET (foos / param<int>(), get_foo_by_id),
    POST(foos / body<foo>() , post_new_foo),
    GET (version / "bars"   , get_list_all_bars)
  );
}
```

```
Response get_list_all_foos(const Request& rq) {
   return {list_all_foos(), OK};
}

Response get_foo_by_id(const Request& rq, int id) {
   auto foo = find_foo(id);
   if (foo) return {*foo, OK};
   return {nullptr, NotFound};
}

Response post_new_foo(const Request& rq,
                      const foo& foo) {
   all_foos.push_back(foo);
   return {foo, Created};
}

Response get_list_all_bars(const Request& rq) {
   return {list_all_bars(), OK};
}
```

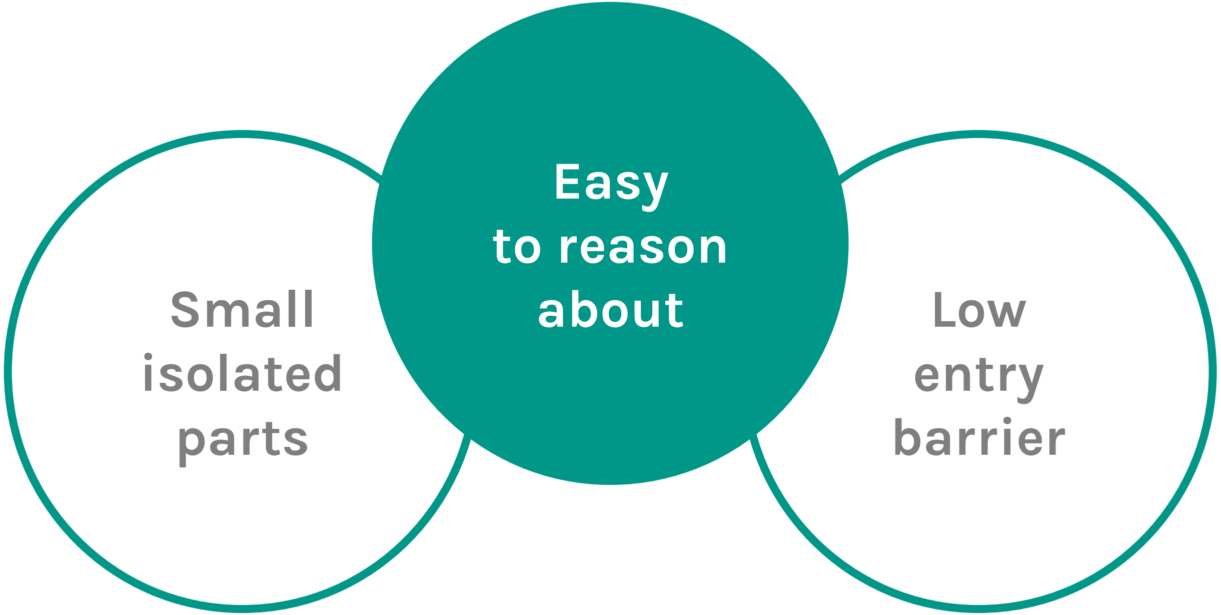
Cohesive router

```
auto api = router(
  GET (foos               , list_all_foos),
  GET (foos / param<int>(), get_foo_by_id),
  POST(foos / body<foo>() , insert_new_foo),
  GET (version / "bars"   , list_all_bars)
);
```

Decoupled handlers

```
Response list_all_foos(const Request& rq);

Response list_all_bars(const Request& rq);

Response get_foo_by_id(const Request& rq, int id);

Response insert_new_foo(const Request& rq, const foo& foo);
```

Client code is simplified

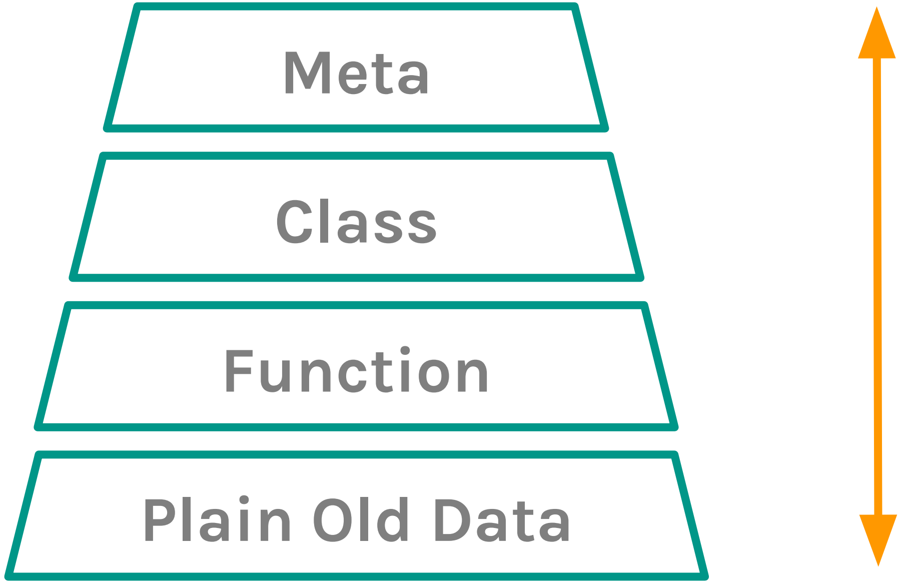


Small isolated parts

Easy to reason about

Low entry barrier

Less [[Arcane]] magic



Meta

Class

Function

Plain Old Data

Powerful
& Complex

vs

Simple &
Composable

5.

Getting DRY

Do not
Repeat
Yourself

> Most people take DRY to mean you shouldn’t duplicate code. That's not its intention.

Dave Thomas

(Defined DRY with Andy Hunt)

“

Every piece of system knowledge should have one authoritative, unambiguous representation.

Dave Thomas

(Defined DRY with Andy Hunt)

DRY is about information

1

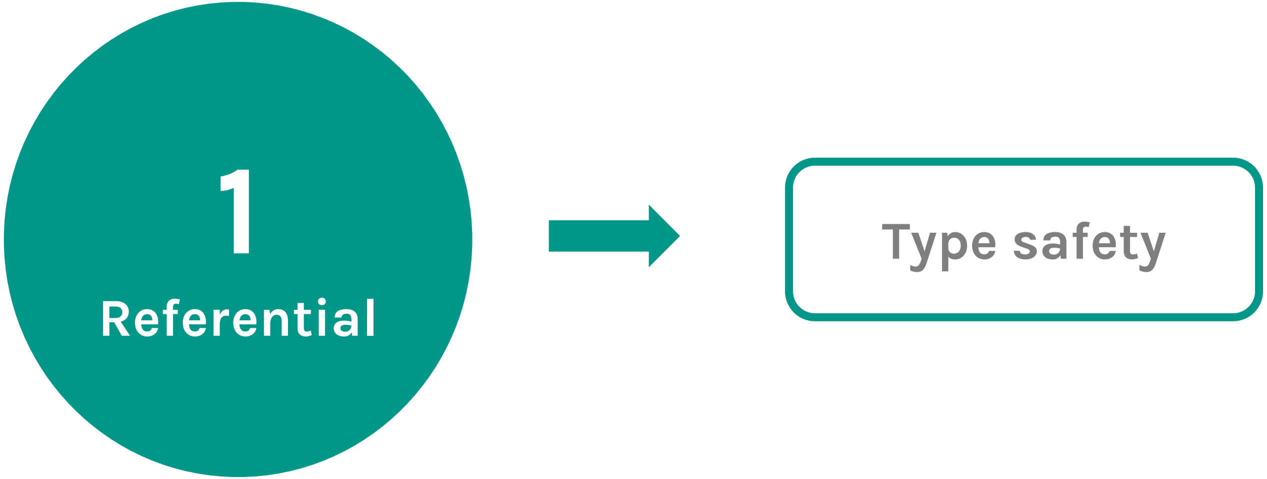
Referential

=

Router

- Code
- Resource
- Pick one

Deriving type safety



1

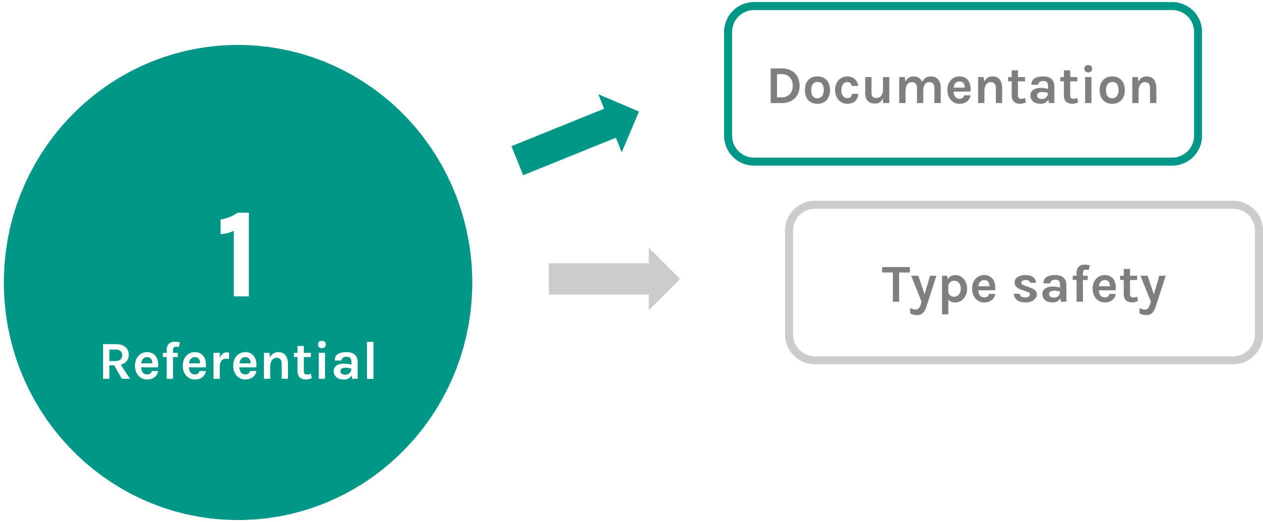
Referential

Type safety

Deriving type safety

```
GET(version / "foos" / param<int>(),
    [](const Request& rq, int id)
    {
      //Implementation
    });
```

Generating documentation



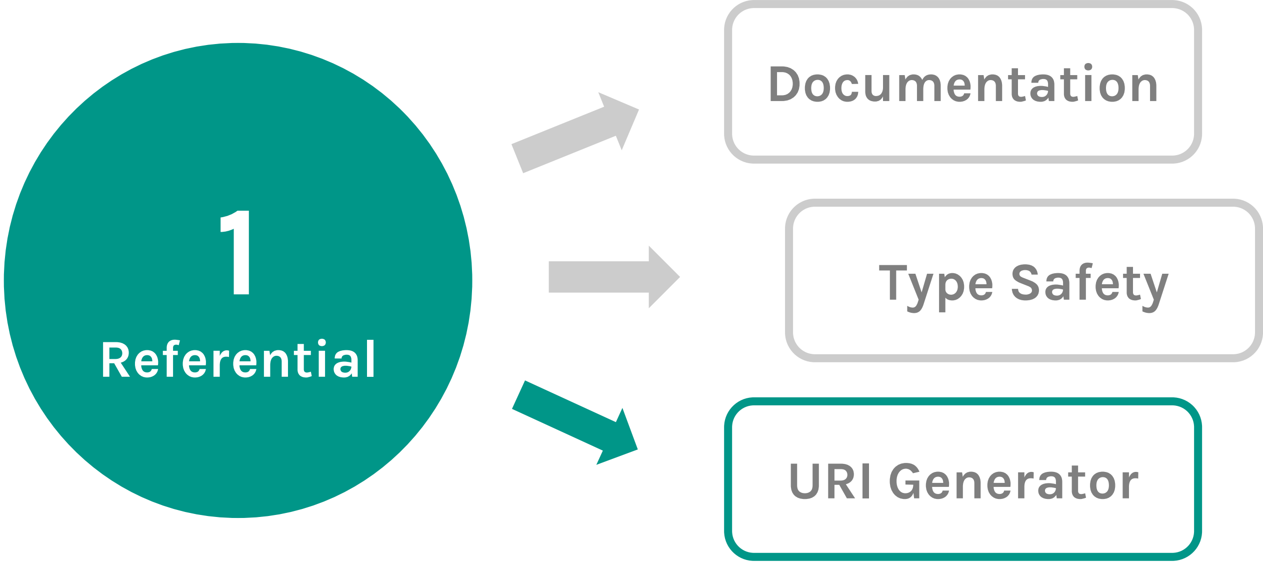
Generating documentation

```
auto api = router(
  GET(foos_path, get_list_all_foos),
  GET(foos_path / param<int>(), get_foo_by_id));

describe(api);
> "GET: v1/foos"
> "GET: v1/foos/([0-9]+)"
```

Generating documentation

```
auto api = router(
  GET(foos_path, get_list_all_foos),
  GET(foos_path / param<int>(), get_foo_by_id));

describe(api);
> "GET: v1/foos"
> "GET: v1/foos/([0-9]+)"
```

Generating random routes



1

Referential

Documentation

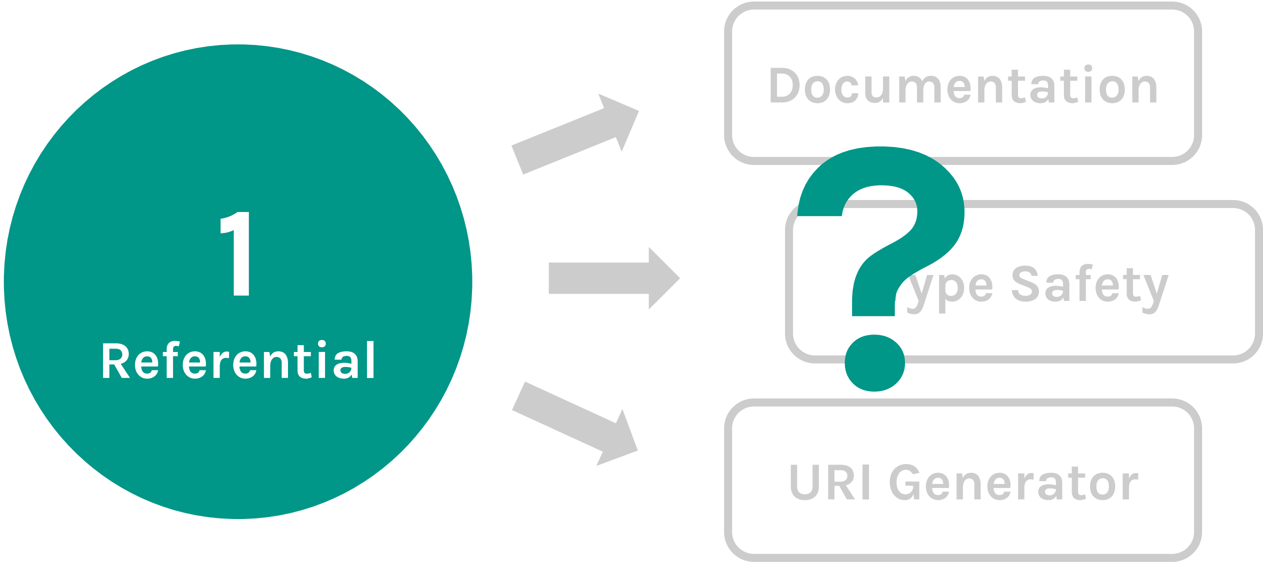
Type Safety

URI Generator

Generating random routes

```
auto api = router(
  GET(foos_path, get_list_all_foos),
  GET(foos_path / param<int>(), get_foo_by_id));

for (auto r: generate(api, 2)) {
   api(r);
}
```

Opening information



1

Referential

Documentation

ype Safety

URI Generator

API as responsible for data

```
auto api = router(
  GET (foos               , list_all_foos),
  GET (foos / param<int>(), get_foo_by_id),
  POST(foos / body<foo>() , insert_new_foo),
  GET (version / "bars"   , list_all_bars)
);
```

6.

Key Takeaways

Elements of Functional Design

Defining concepts

Abstract

=

Precise

- Think functions
- Code vs Meaning
- Contracts, concepts

Composition

Complexity

out of

Simplicity

○ Separate first

○ Compose after

○ Simple constructs

Think about information

Manage

&

Transform

- ○ Plain old data
- ○ Keep it carefully
- ○ Leverage on data

Functional design in C++

Steal ideas

not

Features

○ Haskell is not C++

○ Motivations & ideas

○ Adapt powerful ideas

THANKS!

Any questions?

Follow us at @quduval and @jeremydemeule

@Work_at_Murex

7.

Links & Resources

Links & Resources

- Growing popularity of Spring Boot: http://redmonk.com/fryan/2017/06/22/language-framework-popularity-a-look-at-java-june-2017/

- DRY is not about code duplication: http://www.artima.com/intv/dry.html

Existing solutions (C++)

○ Pistache: https://github.com/oktal/pistache

○ QTTP Server: https://github.com/supamii/QttpServer

Existing solutions (C++)

○ Silicon framework
http://siliconframework.org

○ Elle
https://github.com/infinit/elle

Existing solutions (FP)

- Compojure + Clout (Clojure)
  https://github.com/weavejester/compojure
  https://github.com/weavejester/clout

- Servant (Haskell)
  https://haskell-servant.github.io

- Pedestal (Clojure)
  http://pedestal.io

Aspect Oriented Programming

- https://en.wikipedia.org/wiki/Aspect-oriented_programming

- Example of annotation for AOP (Spring) https://stackoverflow.com/questions/4829088/java-aspect-oriented-programming-with-annotations