A typesafe API

With Servant

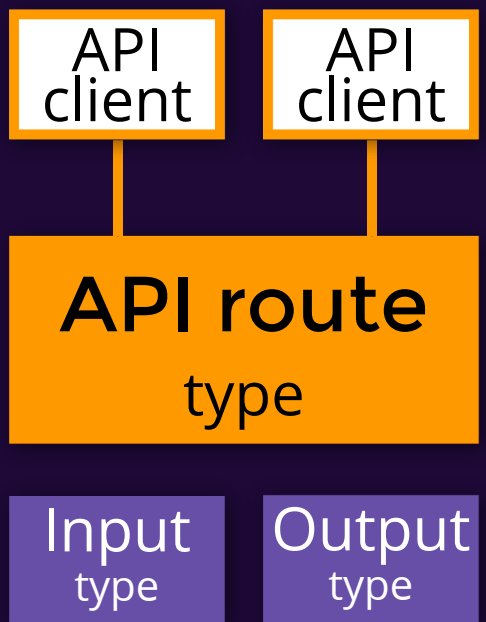
Caroline GAUDREAU

(@akhesacaro)

fretlink

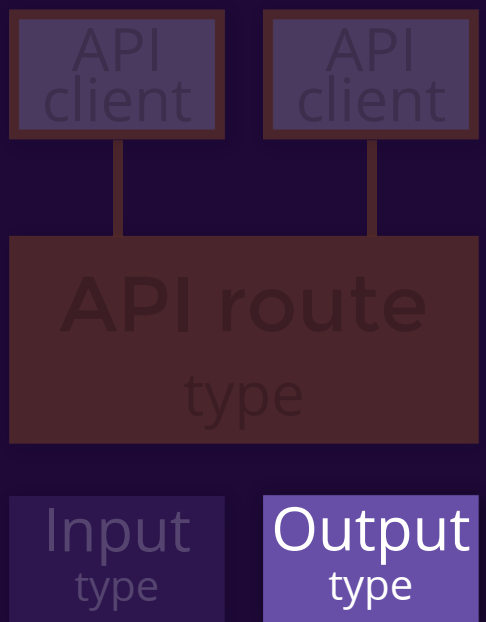
A Typesafe API

With Servant



A Typesafe API

With Servant



Movies Catalog

Se7en

Crime, Drama, Mystery

1995

Minority Report

Action, Crime, Mystery

2002

Types in Haskell

First step : defining domain types

Sum Type

Type with an **OR**

```haskell
data Genre = Horror | Anime | Action | Family | Crime | Drama | Mystery
```

Types in Haskell

First step : defining domain types

Sum Type

Type with an **OR**

```haskell
data Genre = Horror | Anime | Action | Family | Crime | Drama | Mystery
```

```haskell
getMoviesByGenre :: Genre -> [Movie]
getMoviesByGenre Horror = ...
getMoviesByGenre Action = ...
```

What about the missing genres?

Types in Haskell

First step : defining domain types

Sum Type

Type with an **OR**

```haskell
data Genre = Horror | Anime | Action | Family | Crime | Drama | Mystery
```

```haskell
getMoviesByGenre :: Genre -> [Movie]
getMoviesByGenre Horror = ...
getMoviesByGenre Action = ...
```

What about the missing genres?

```
/home/akhesacaro/akhesa/confs/typesafe_api_servan
    Pattern match(es) are non-exhaustive
    In an equation for 'getMoviesByGenre':
        Patterns not matched:
            Anime
            Family
            Crime
            Drama
            ...
```

Types in Haskell

First step : defining domain types

Sum Type

Type with an **OR**

```haskell
data Genre = Horror | Anime | Action | Family | Crime | Drama | Mystery
```

Types — Values

```haskell
myFavoriteGenre :: Genre
myFavoriteGenre = Crime
```

Types in Haskell

First step : defining domain types

Records (Product type)

type with attributes

```haskell
data Movie = Movie { title   :: String
                   , genre :: [Genre]
                   , year :: Int
                   }
```

Types in Haskell

First step : defining the domain's types

Records

type with attributes

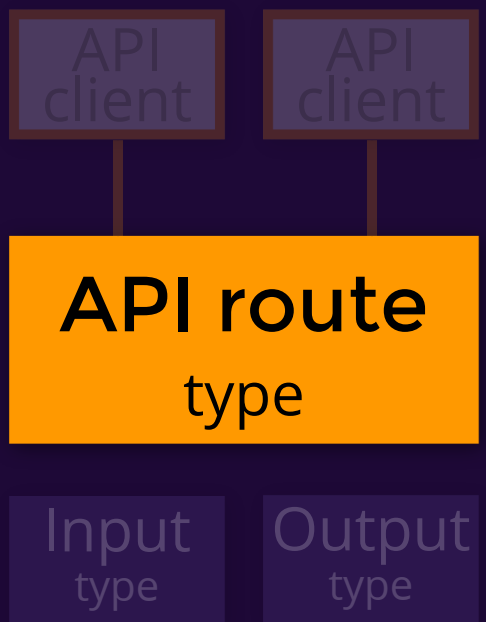
```
1  data Movie = Movie { title    :: String
2                     , genre :: [Genre]
3                     , year :: Int
4                     }
```

Types

Values

A Typesafe API

With Servant



JSON

API description

The API IS a type

```
http://localhost:8081/content?sortby=ByGenre?year=2000
```

API description

The API IS a type

```
http://localhost:8081/content?sortby=ByGenre?year=2000
```

```haskell
data SortBy = ByTitle | ByGenre
```

```haskell
type Year = Int
```

API description

The API IS a type

```
http://localhost:8081/content?sortby=ByGenre?year=2000
```

```haskell
data SortBy = ByTitle | ByGenre
```

```haskell
type Year = Int
```

```
/ → :>
```

API description

The API IS a type

`http://localhost:8081/content?sortby=ByGenre?year=2000`

```
1 data SortBy = ByTitle | ByGenre
```

```
1 type Year = Int
```

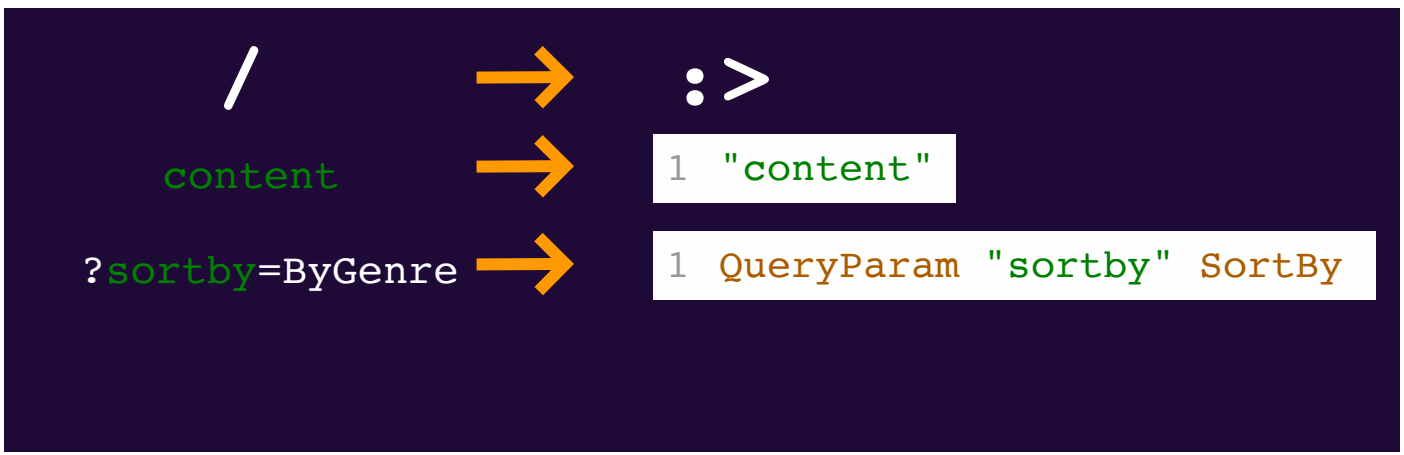
API description

The API IS a type

```
http://localhost:8081/content?sortby=ByGenre?year=2000
```

```haskell
data SortBy = ByTitle | ByGenre
```

```haskell
type Year = Int
```



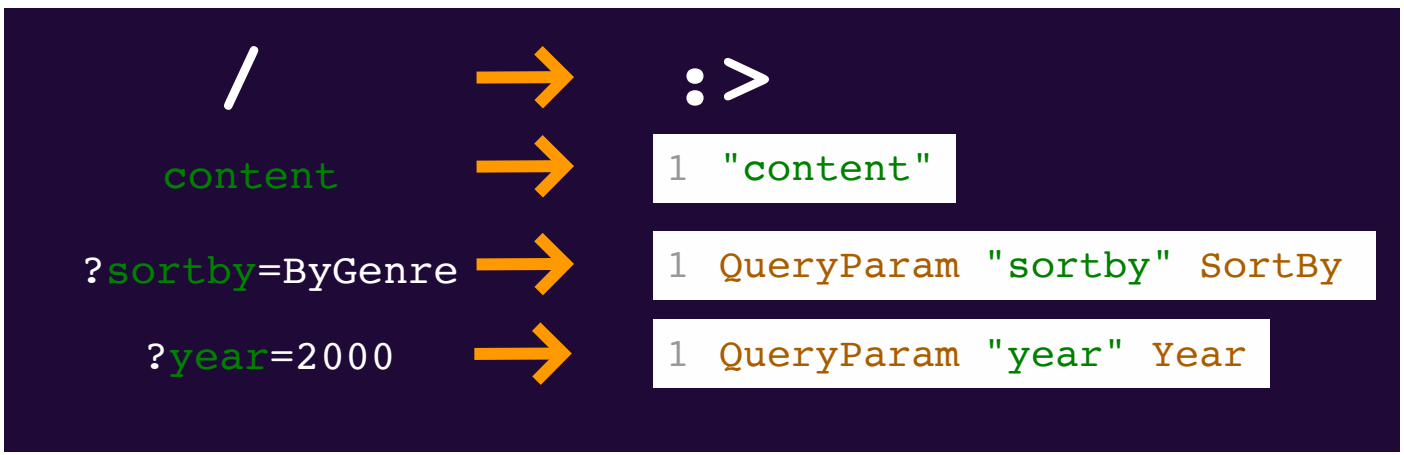
API description

The API IS a type

`http://localhost:8081/content?sortby=ByGenre?year=2000`

```
1 data SortBy = ByTitle | ByGenre
```

```
1 type Year = Int
```



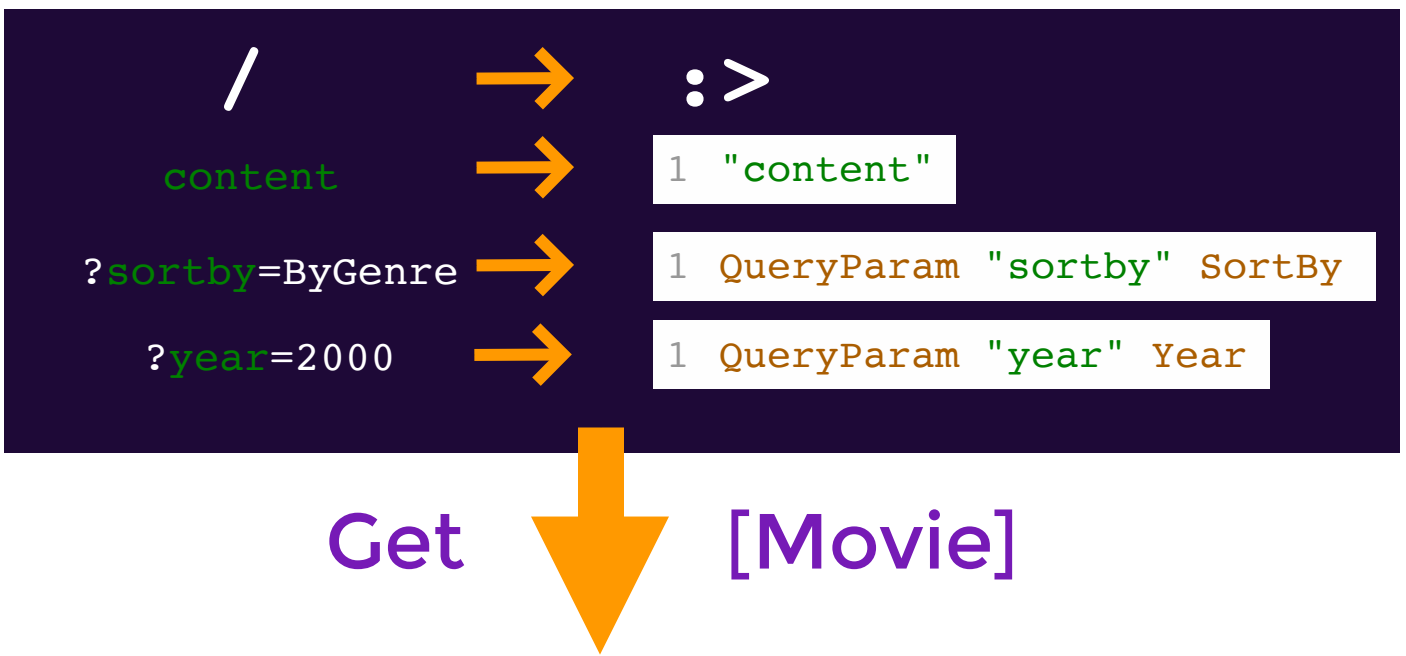
API description

The API IS a type

http://localhost:8081/content?sortby=ByGenre?year=2000

```
1 data SortBy = ByTitle | ByGenre
```

```
1 type Year = Int
```




```
1 "content" :> QueryParam "sortby" SortBy :> QueryParam "year" Year :> Get '[JSON] [Movie]
```

Wrong catch !?

What if a client call with an inexistant sortby value?

```haskell
data SortBy = ByTitle | ByGenre
```

http://localhost:8081/content?sortby=**BugsBunny**

```haskell
"content"
:> QueryParam "sortby" SortBy
:> QueryParam "year" Year
:> Get '[JSON] [Movie]
```

Wrong Query Param Output

Maybe

A world without Null and Undefined

Maybe Something is either **Nothing** or **Just Something**

```
1 data Maybe a = Nothing | Just a
```

Maybe

A world without Null and Undefined

Maybe Something is either **Nothing** or **Just Something**

```
1 data Maybe a = Nothing | Just a
```

Example :

```
1 sayMyName :: Bool -> Maybe String
2 sayMyName False = Nothing
3 sayMyName True = Just "Heisenberg"
```

Maybe String is either **Nothing** or **Just String**

Function

We have to make an Handler

Get all contents

May filter by year And/Or

May sort by title or genre

Function

We have to make an Handler

Get all contents

May filter by year And/Or

May sort by title or genre

Last type
=
Result type

```
1 getContentHandler :: Maybe SortBy -> Maybe Year -> Handler [Movie]
```

Type parameters

Function

We have to make an Handler

Get all contents

May filter by year And/Or

May sort by title or genre

Last type
=
Result type

```
1 getContentHandler :: Maybe SortBy -> Maybe Year -> Handler [Movie]
```

Type parameters

```
1 getContentHandler Nothing         Nothing      = ..
2 getContentHandler Nothing         (Just year)  = ..
3 getContentsHandler (Just sortBy)  Nothing      = ..
4 getContentHandler (Just sortBy)   (Just year)  = ..
```

Type safety?

How does Servant verify type safety?

`http://localhost:8081/content?sortby=ByGenre?year=2000`

API type

```
type CatalogAPI =

  "content" :> QueryParam "sortby" SortBy :> QueryParam "year" Year :> Get '[JSON] [Movie]
```

Handler

```
getContentHandler :: Maybe SortBy -> Maybe Year -> Handler [Movie]
```

Servant confronts declared **Types** combinations to the **Values** combinations

How servant works?

How does Servant verify type safety?

API type

```haskell
type CatalogAPI =

  "content" :> QueryParam "sortby" SortBy :> QueryParam "year" Year :> Get '[JSON] [Movie]
```

Handler

```haskell
getContentHandler :: Maybe SortBy          -> Maybe Year          -> Handler [Movie]
```

Server

```haskell
server :: Server CatalogAPI

server = getContentHandler
```

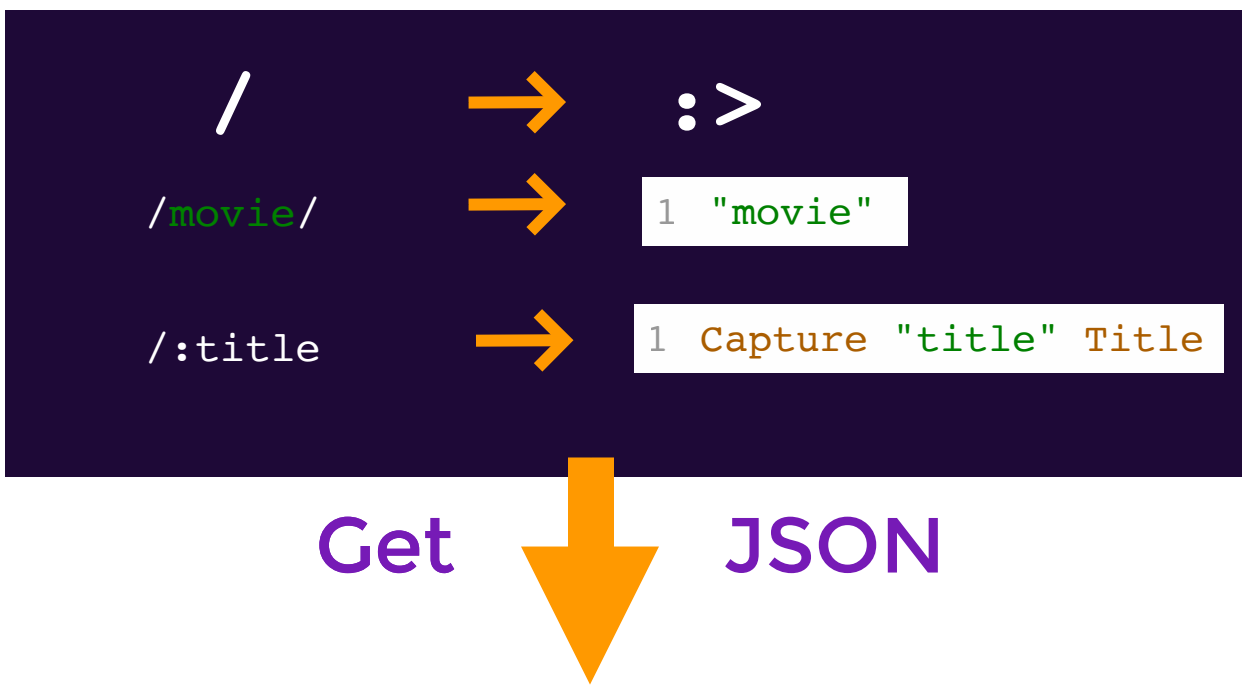
Many endpoints ?

HOWTO

Many endpoints ?

Let's add one

`http://localhost:8081/movie/:title`




```
1 "movie" :> Capture "title" Title :> Get JSON (Maybe Movie)
```

How to combine?

With our CatalogAPI (world of types)?

http://localhost:8081/movie/:title

API type

```
1 type CatalogAPI =
2   "content" :> QueryParam "sortby" SortBy :> QueryParam "year" Year :> Get '[JSON] [Movie]
3
4   :<|> "movie" :> Capture "title" Title :> Get JSON (Maybe Movie)
```

How to combine?

With our CatalogAPI (world of types)?

http://localhost:8081/movie/:title

API type

```
1 type CatalogAPI =
2   "content" :> QueryParam "sortby" SortBy :> QueryParam "year" Year :> Get '[JSON] [Movie]
3
4   :<|> "movie" :> Capture "title" Title :> Get JSON (Maybe Movie)
```

```
1 getMovieHandler :: Title -> Handler (Maybe Movie)
```

Handler

How to combine?

With our server (world of values)?

http://localhost:8081/movie/:title

API type

```haskell
type CatalogAPI =
  "content" :> QueryParam "sortby" SortBy :> QueryParam "year" Year :> Get '[JSON] [Movie]

  :<|> "movie" :> Capture "title" Title :> Get JSON (Maybe Movie)
```

Handler

```haskell
getMovieHandler :: Title -> Handler (Maybe Movie)
```

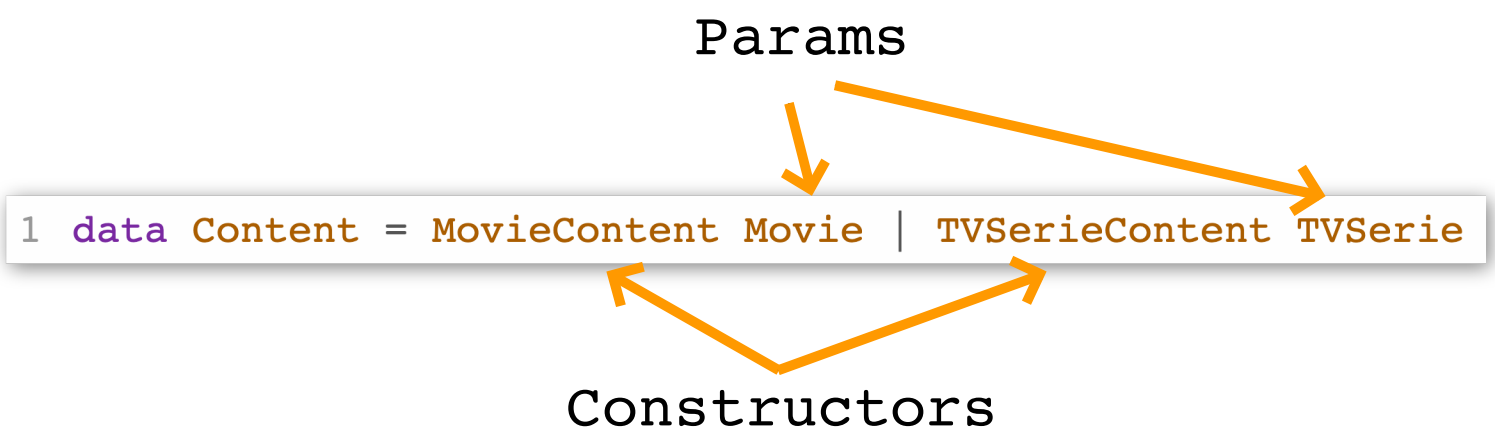
Server

```haskell
server :: Server CatalogAPI

server = getMoviesHandler :<|> getMovieHandler
```

OR

We want Movies OR TV Series

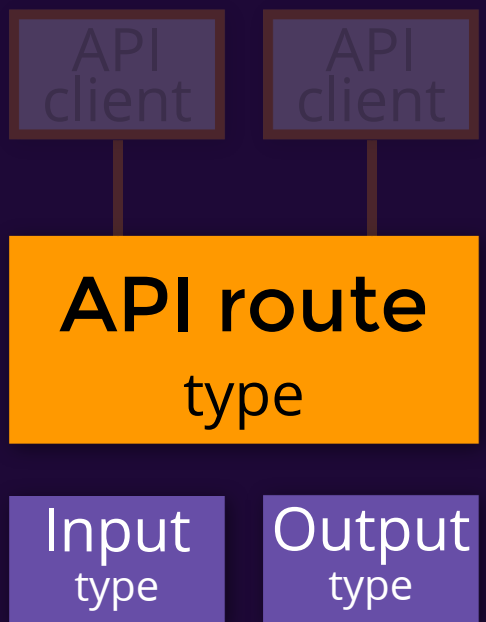



```haskell
data TVSerie =
    TVSerie { title :: Title
            , genre :: [Genre]
            , seasonNb :: Int
            , year :: Year
            }
```

Change API type
DEMO

A Typesafe API

With Servant



JSON

Type class

Kind of capabilities

When a **type** "instantiates" a **type class**

this means that the type **is capable of doing** ....

```haskell
data Movie = Movie { title :: Title
                   , genre :: [Genre]
                   , year :; Int
                   }
                   deriving stock (Show, Generic)
                   deriving anyclass (ToJSON)
```

ToJSON type class

Movie is capable of be transformed **to** a **JSON**

Type class

Kind of capabilities

ToJSON type class

Transform **your type into an Aeson type**

```haskell
class ToJSON a where

    toJSON     :: a -> Value
```

functions

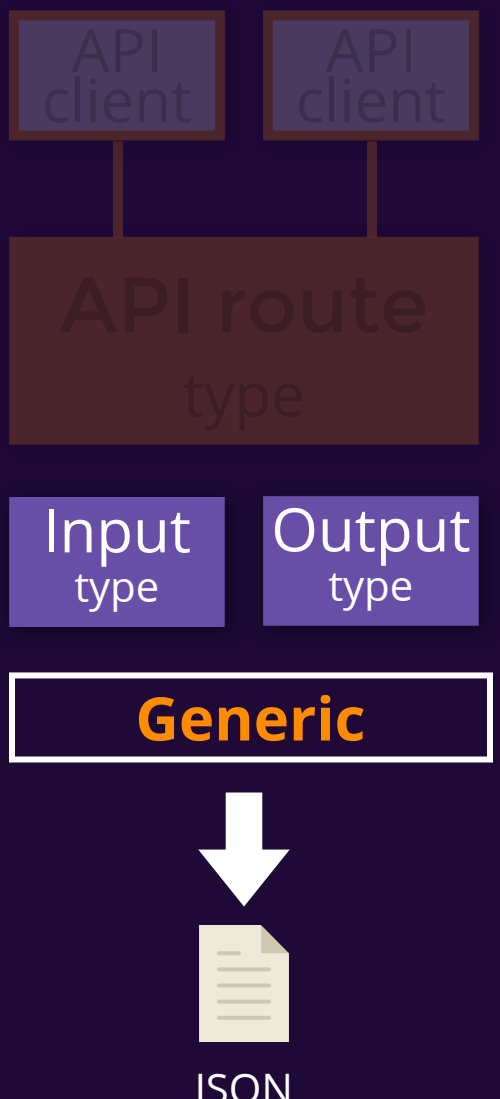
```haskell
toEncoding :: a -> Encoding

toJSONList :: [a] -> Value

toEncodingList :: [a] -> Encoding
```

A Typesafe API

With Servant



Generic

under the hood

Polymorphism

Parametric, Ad-hoc, Structural

Parametric polymorphism

One **definition** for **every possible type**

```
1 fst :: (a,a) -> a
```

Polymorphism

Parametric, Ad-hoc, Structural

Parametric polymorphism

One **definition** for **every possible type**

```
1 fst :: (a,a) -> a
```

Ad-hoc polymorphism

Different implementation for **every possible type**

```
1 class Show a where
2         show :: a -> String
3
4 instance (Maybe a) Show
5 instance (Either a) Show
6 instance Bool Show
```

Polymorphism

Parametric, Ad-hoc, Structural

Parametric polymorphism

One **definition** for **every possible type**

```
1 fst :: (a,a) -> a
```

Ad-hoc polymorphism

Different implementation for **every possible type**

```
1 class Show a where
2         show :: a -> String
3
4 instance (Maybe a) Show
5 instance (Either a) Show
6 instance Bool Show
```

Structural polymorphism

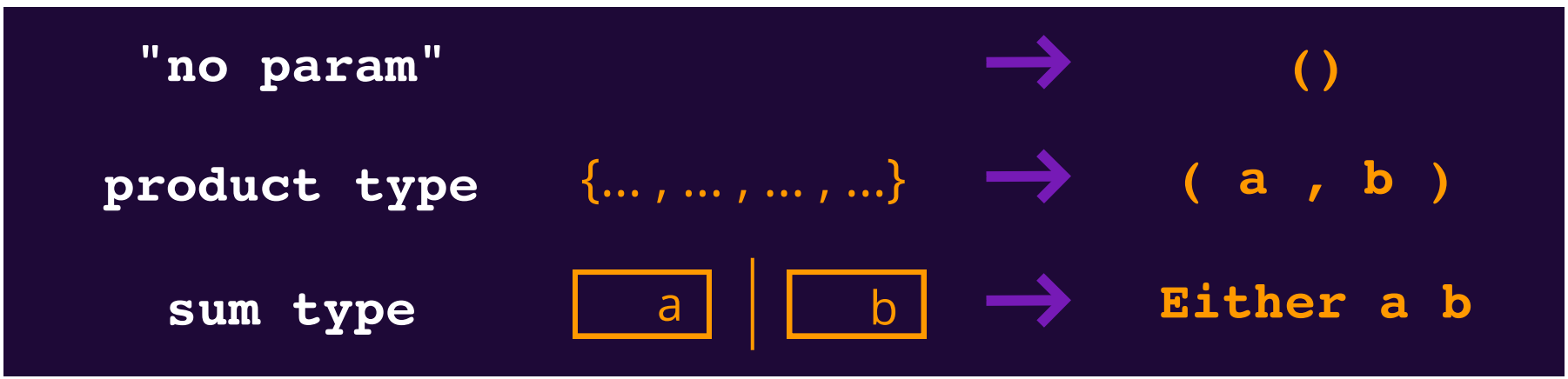
Different implementation for **every canonical representation**

```
1 data Maybe a     = Nothing | Just a
2 data PeutEtre a = Rien     | Juste a
```

Canonical representation

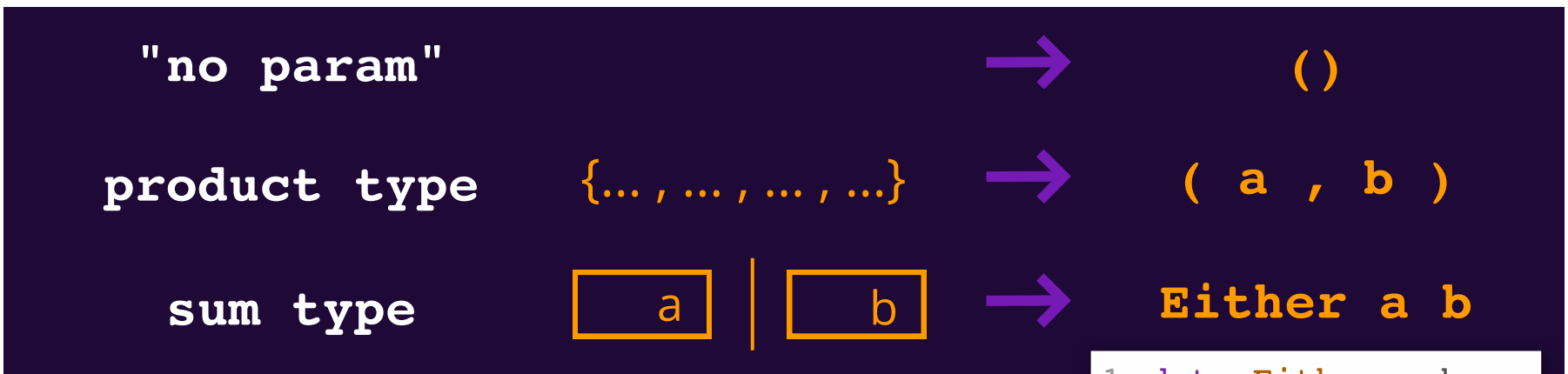
Structural Polymorphism

Highly regular and predictable



Structural Polymorphism

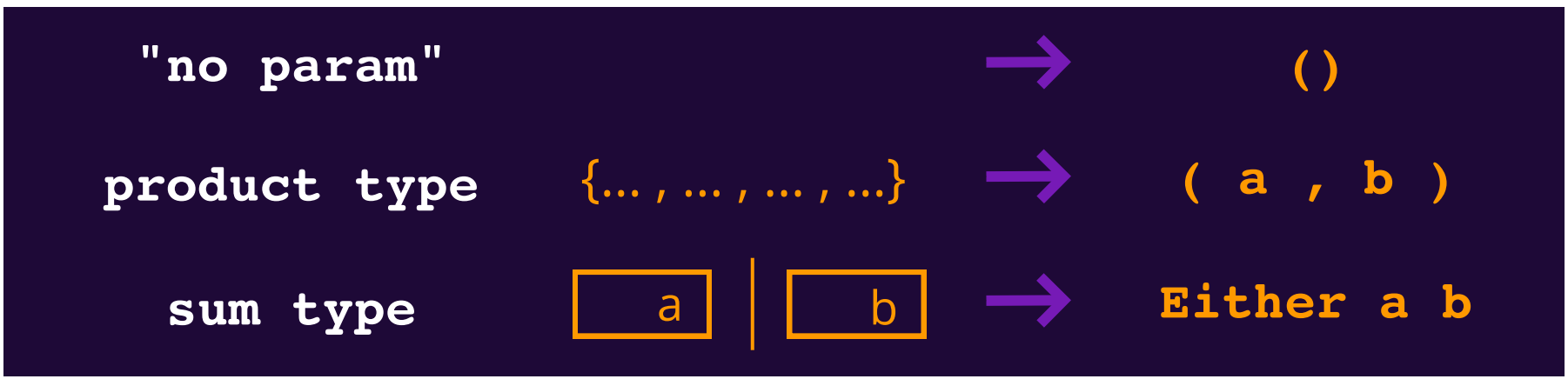
Highly regular and predictable




```haskell
data Either a b =
  Left a
  Right b
```

Structural Polymorphism

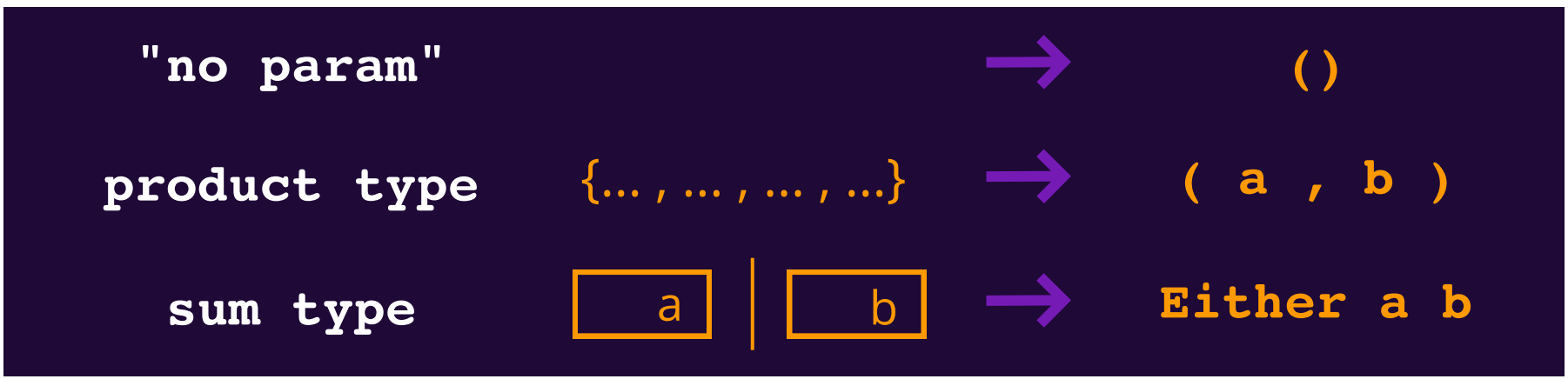
Highly regular and predictable




```
1 data PaymentMethod = Card CardId | Check CheckId
```

Structural Polymorphism

Highly regular and predictable

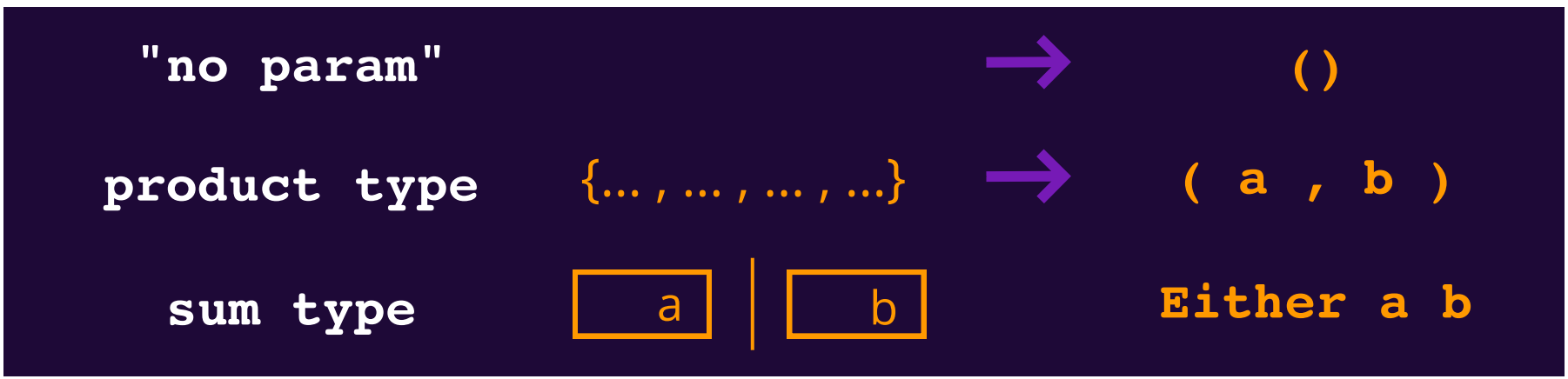



```
1 data PaymentMethod = Card CardId | Check CheckId
```

```
1 Either CardId CheckId
```

Structural Polymorphism

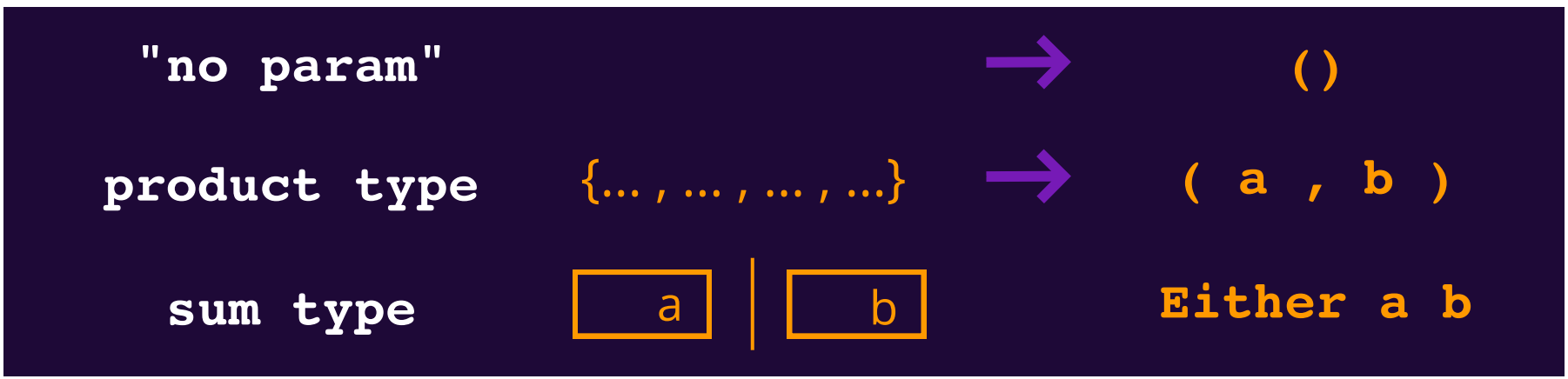
Highly regular and predictable




```
1 data Maybe a = Nothing | Just a
```

Structural Polymorphism

Highly regular and predictable

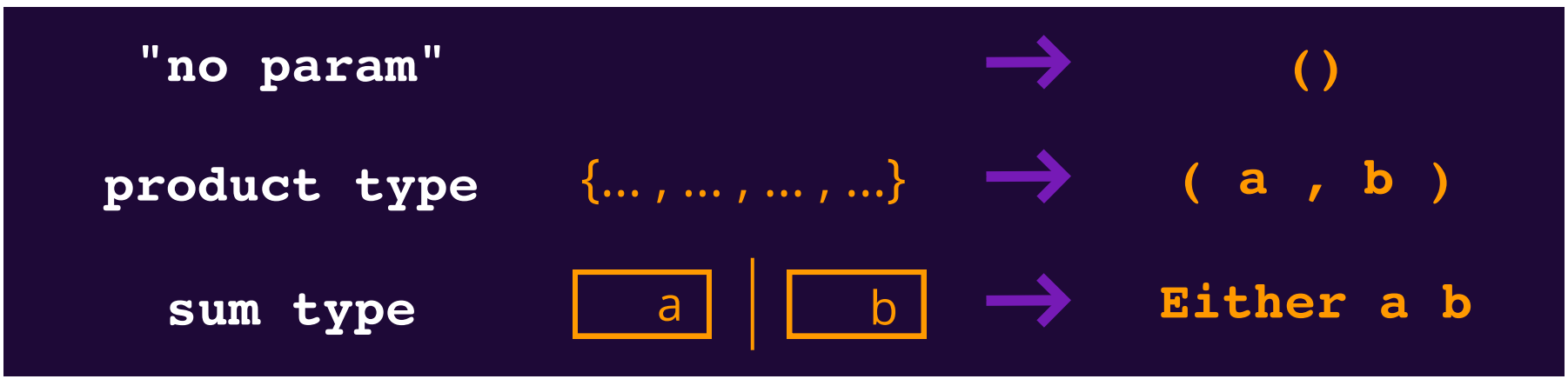



```
1 data Maybe a = Nothing | Just a
```

```
1 Either () a
```

Structural Polymorphism

Highly regular and predictable




```
1 data Movie = Movie
2 { title :: Title
3 , min :: Duration }
```

Structural Polymorphism

Highly regular and predictable

"no param"		→	()
product type	{… , … , … , …}	→	(a , b)
sum type	a | b	→	Either a b

```
1 data Movie = Movie
2 { title :: Title
3 , min :: Duration }
```

```
1 ( Title , Duration )
```

Structural Polymorphism

Highly regular and predictable


```
1 data Content = MovieContent Movie | TVSerieContent TVSerie
```

```
1 Either Movie TVSerie
```

Structural Polymorphism

Highly regular and predictable


```haskell
data Content = MovieContent Movie | TVSerieContent TVSerie
```

```haskell
data Movie = Movie
{ title :: Title
, min :: Duration }
```

```haskell
Either Movie TVSerie
```

```haskell
data TVSerie = TVSerie
{ title :: Title
, seasonsNb :: Int }
```

Structural Polymorphism

Highly regular and predictable


```
1 data Content = MovieContent Movie | TVSerieContent TVSerie
```

```
1 Either Movie TVSerie
```

```
1 data Movie = Movie
2 { title :: Title
3 , min :: Duration }
```

```
1 data TVSerie = TVSerie
2 { title :: Title
3 , seasonsNb :: Int }
```

```
1 Either (Title , Duration) (Title, Int)
```

Structural Polymorphism

Why is it reliable?

Because it is **Small** and **Simple**

```
1 map :: (a -> b) -> [a] -> [b]
2 map [] = []
3 map (x:xs) = (f x):xs
```

Like a map implementation

Structural Polymorphism

Why is it predictable?

implemented with **Ad-hoc polymorphism**

Movie	**TV Serie**	**Genre**	**Maybe a**	**...**
⬇	⬇	⬇	⬇	⬇
JSON	**JSON**	**JSON**	**JSON**	**JSON**

A lot of boilerplate
A lot of cases
High probability of errors/divergences

Structural Polymorphism

Why is it predictable?

implemented with **Structural polymorphism**

Product type	**Sum type**	**Value**
⬇	⬇	⬇
JSON	**JSON**	
{ "<name>":<value>,...}	"<constructor>"	"<value>"

Declarative
Few cases
Predictable

Generic

Representation

Based on the **Representation** of the **Type**

Canonical

representation

Generic

Representation

Based on the **Representation** of the **Type**

Canonical

representation

Type

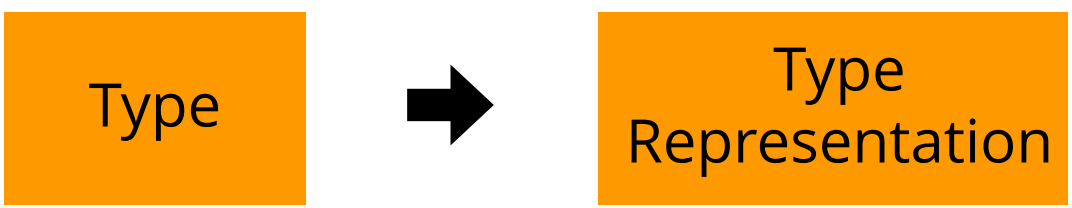
Generic

Representation

Based on the **Representation** of the **Type**

Canonical

representation



Generic

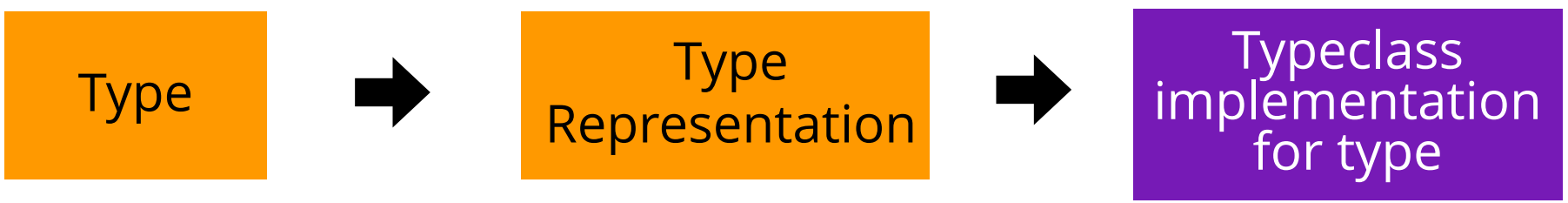
Generic

Representation

Based on the **Representation** of the **Type**

Canonical

representation

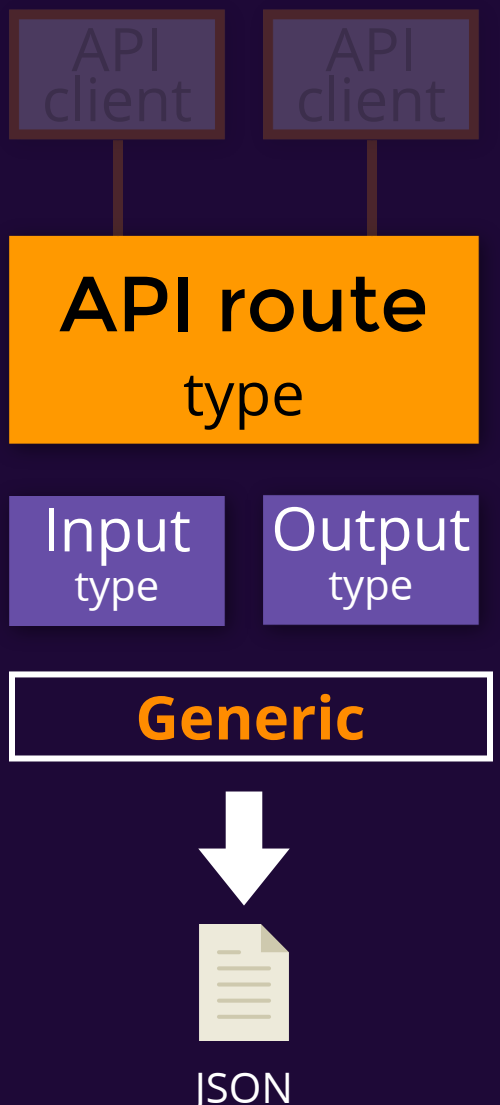


Generic

Typeclass Generic Implementation

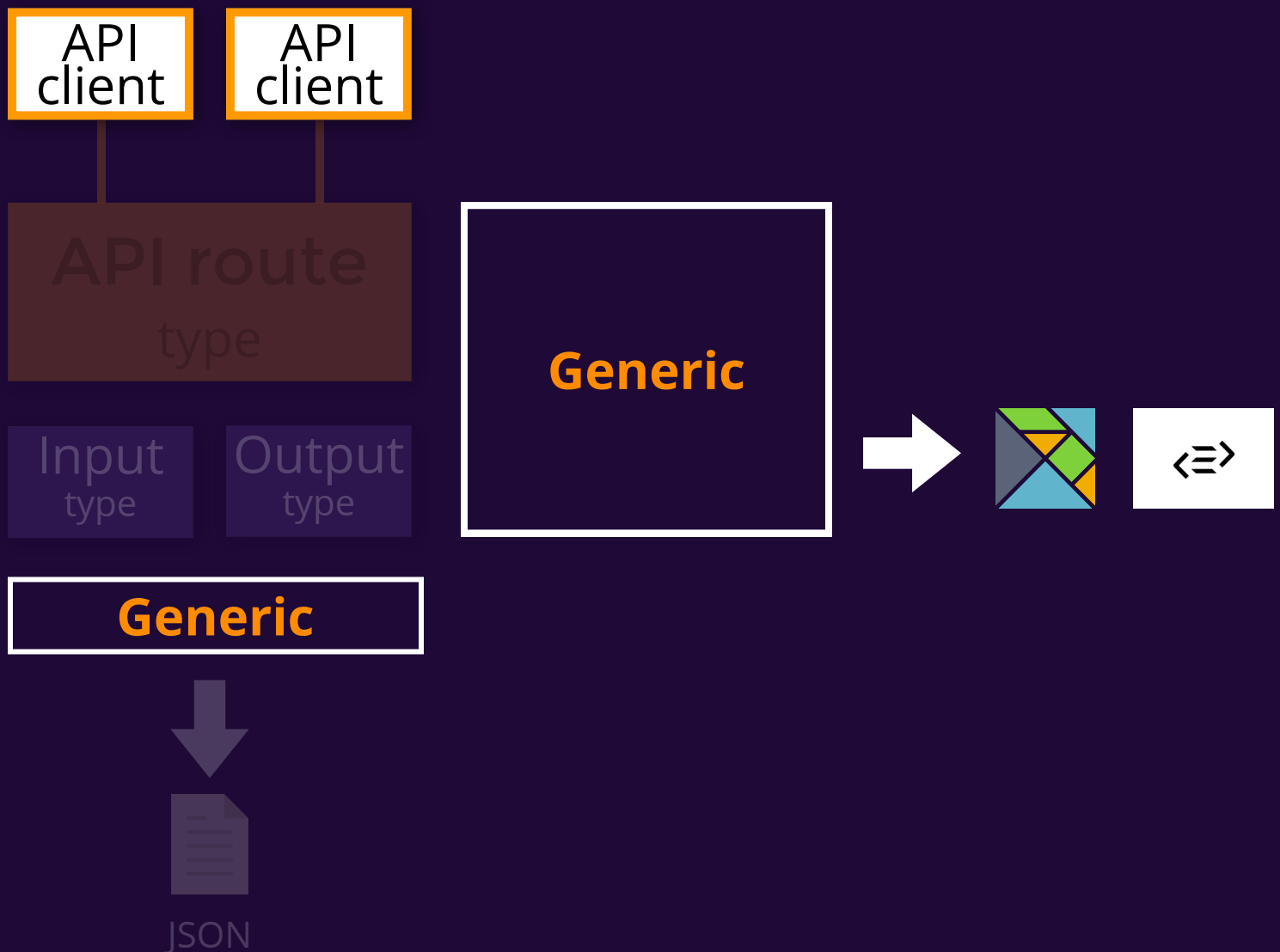
A Typesafe API

With Servant



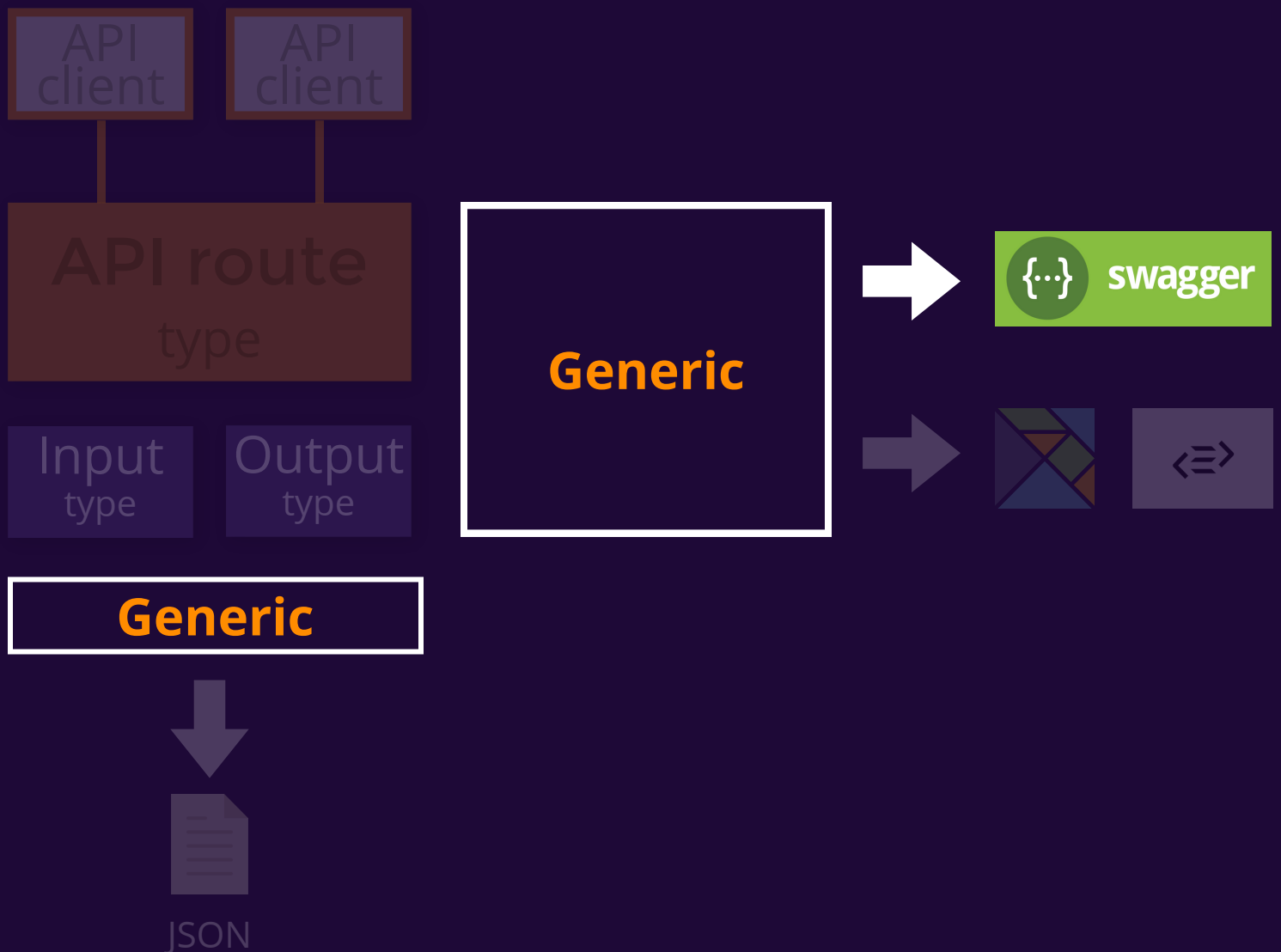
A Typesafe API

With Servant



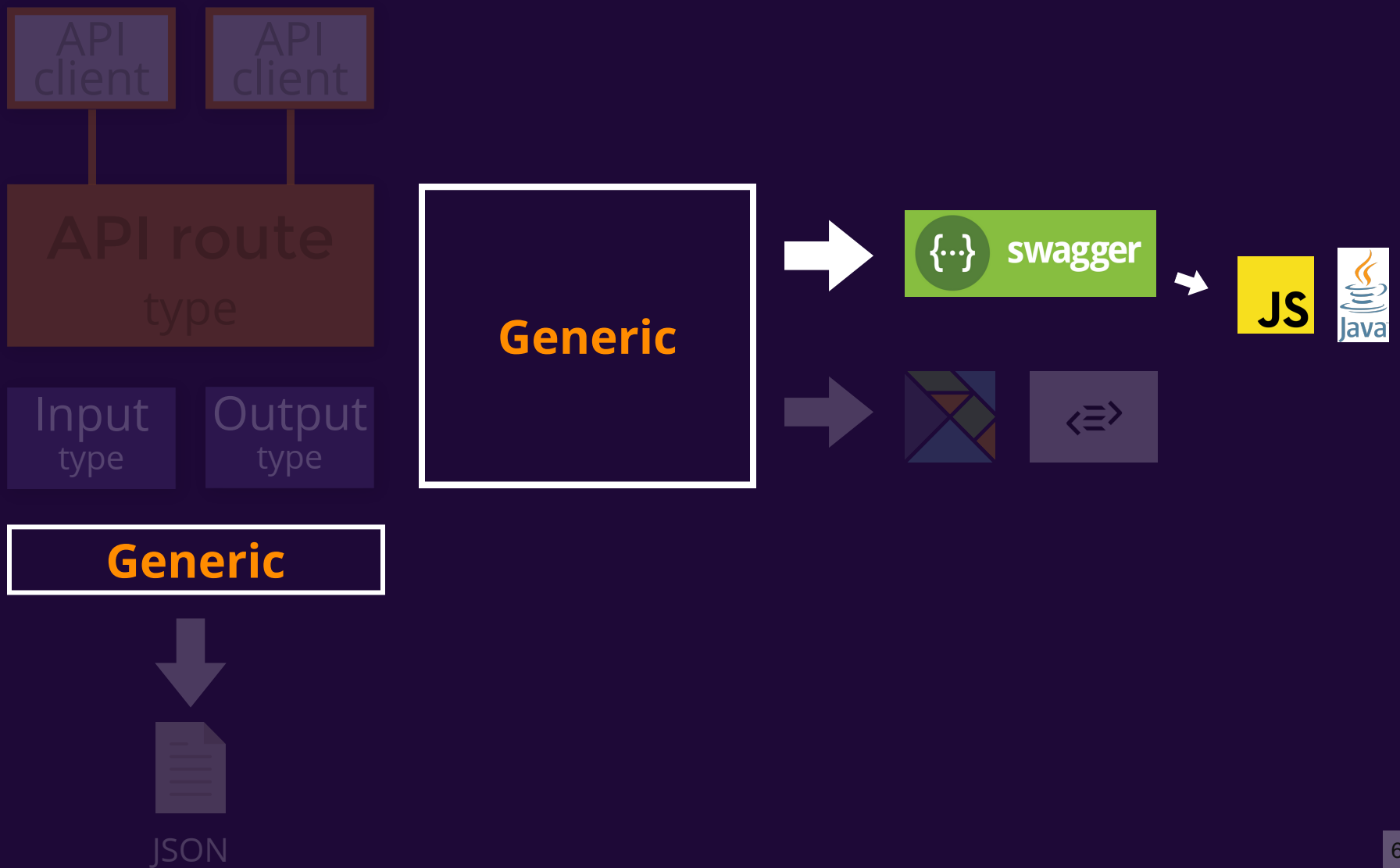
A Typesafe API

With Servant



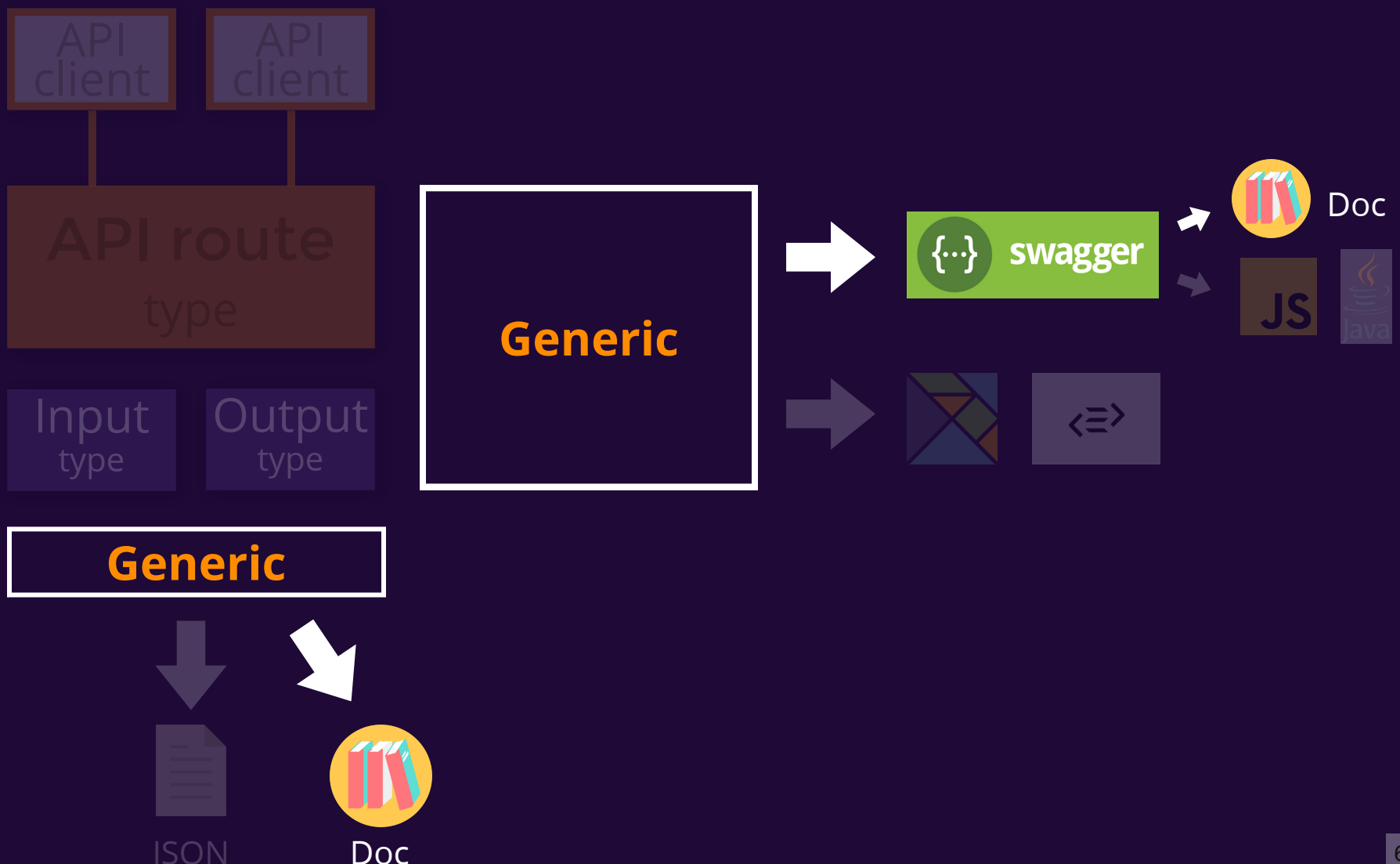
A Typesafe API

With Servant



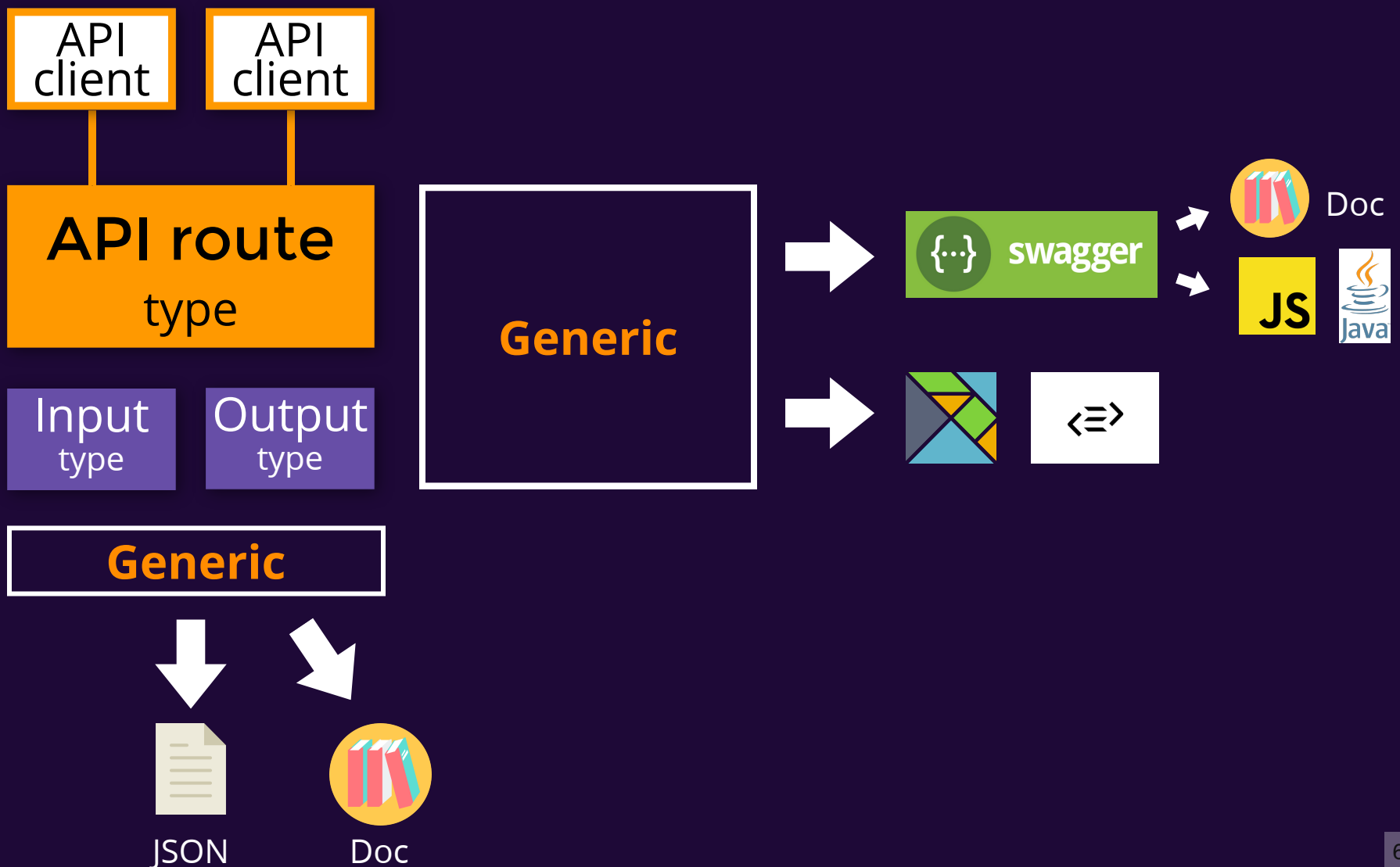
A Typesafe API

With Servant



A Typesafe API

With Servant



Servant other

cool features

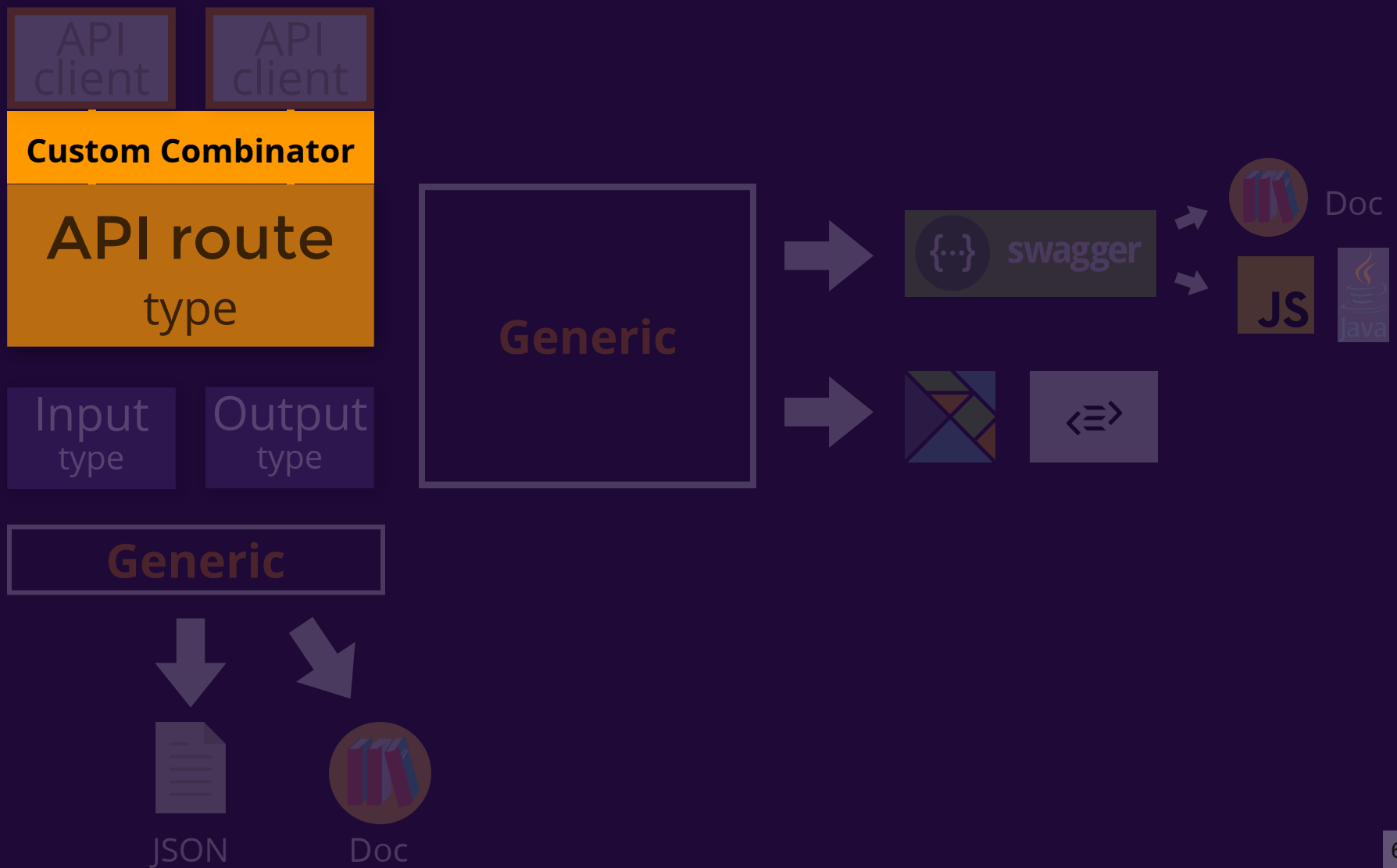
Cool other Servant features

Require Header

```
1 type CatalogAPI =
2
3   "content" :> QueryParam "sortby" SortBy :> QueryParam "year" Year :> Get '[JSON] [Movie]
4
5   :<|>
6     Header' '[Required, Strict] "Authorization" Token :>
7     "movie" :> Capture "title" Title :> Get '[JSON] (Maybe Movie)
```

A Typesafe API

With Servant



Custom combinator

User authentification

Add your own combinator

```
1 type UserAPI = AuthUsers :> "username" :> Get '[JSON] String
```

Expression problem

Philip Wadler, 12 november 1998

```
The Expression Problem is a new name for an old
problem. The goal is to define a datatype by
cases, where one can add new cases to the
datatype and new functions over the datatype,
without recompiling existing code, and while
retaining static type safety (e.g., no casts).
```

How to make a package allowing to extend it (by adding types or functions) without making a new version of that package?

Custom combinator

User authentification

Add your own combinator

```
1 type UserAPI = AuthUsers :> "username" :> Get '[JSON] String
```

Client

Request headers

AuthUser: login;pass

Server

Context

```
1 (Username, Pass)
```

Custom combinator

User authentification

Add your own combinator

```
1 type UserAPI = AuthUsers :> "username" :> Get '[JSON] String
```

Client

Request headers

AuthUser: login;pass

```
1 type UsersContext = (Username, Pass)
```

Server

Context

```
1 (Username, Pass)
```

```
1 data UserInfo =
2   UserInfo {login :: Text
3             , pass :: Text}
```

HasServer

Typeclass

```haskell
class HasServer api context where

  type ServerT api (m :: * -> *) :: *


  route ::
       Proxy api
    -> Context context
    -> Delayed env (Server api)
    -> Router env


  hoistServerWithContext
      :: Proxy api
      -> Proxy context
      -> (forall x. m x -> n x)
      -> ServerT api m
      -> ServerT api n
```

HasServer

Typeclass

Type correspondence

```
type ServerT api (m :: * -> *) :: *
```

HasServer

Typeclass

Type correspondence

```
type ServerT api (m :: * -> *) :: *
```

```
1 type ServerT (AuthUsers :> api) m = Username -> ServerT api m
```

API type

Handler type

HasServer

Typeclass

Route function

```
route :: Proxy api -> Context context -> Delayed env (Server api) -> Router env
```

Fetch Context

Fetch AuthUser Header

DelayedIO -> Error HTTP

HasServer

Typeclass

Route function

```
route :: Proxy api -> Context context -> Delayed env (Server api) -> Router env
```

Fetch Context

Fetch AuthUser Header

DelayedIO -> Error HTTP

```
1 route :: Proxy ( AuthUsers :> api )
2         -> Context (UsersContext ': context)
3         -> Delayed env ( Server (AuthUsers :> api ))
4         -> Router env
```

HasServer

Typeclass

`Delayed`

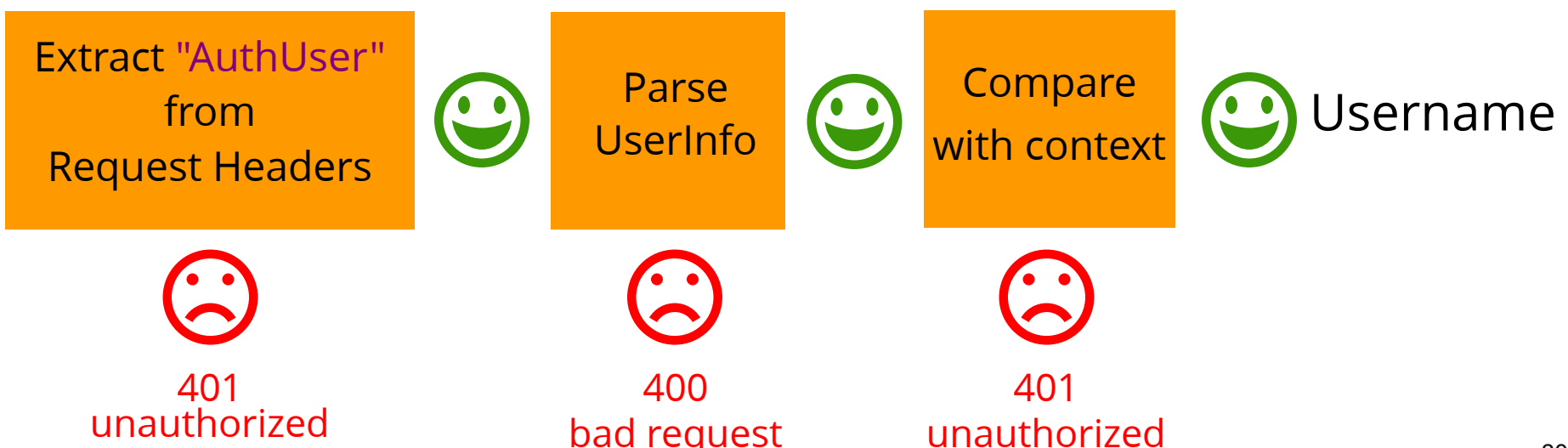
```
1 delayedFailFatal :: ServerError -> DelayedIO a
```

HasServer

Typeclass

`Delayed`

```
1 delayedFailFatal :: ServerError -> DelayedIO a
```



HasServer

Typeclass

Monad function

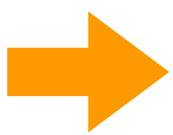
```
hoistServerWithContext :: Proxy api -> Proxy context -> (forall x. m x -> n x) -> ServerT
api m -> ServerT api n
```

HasServer

Typeclass

Monad function

```
hoistServerWithContext :: Proxy api -> Proxy context -> (forall x. m x -> n x) -> ServerT
api m -> ServerT api n
```

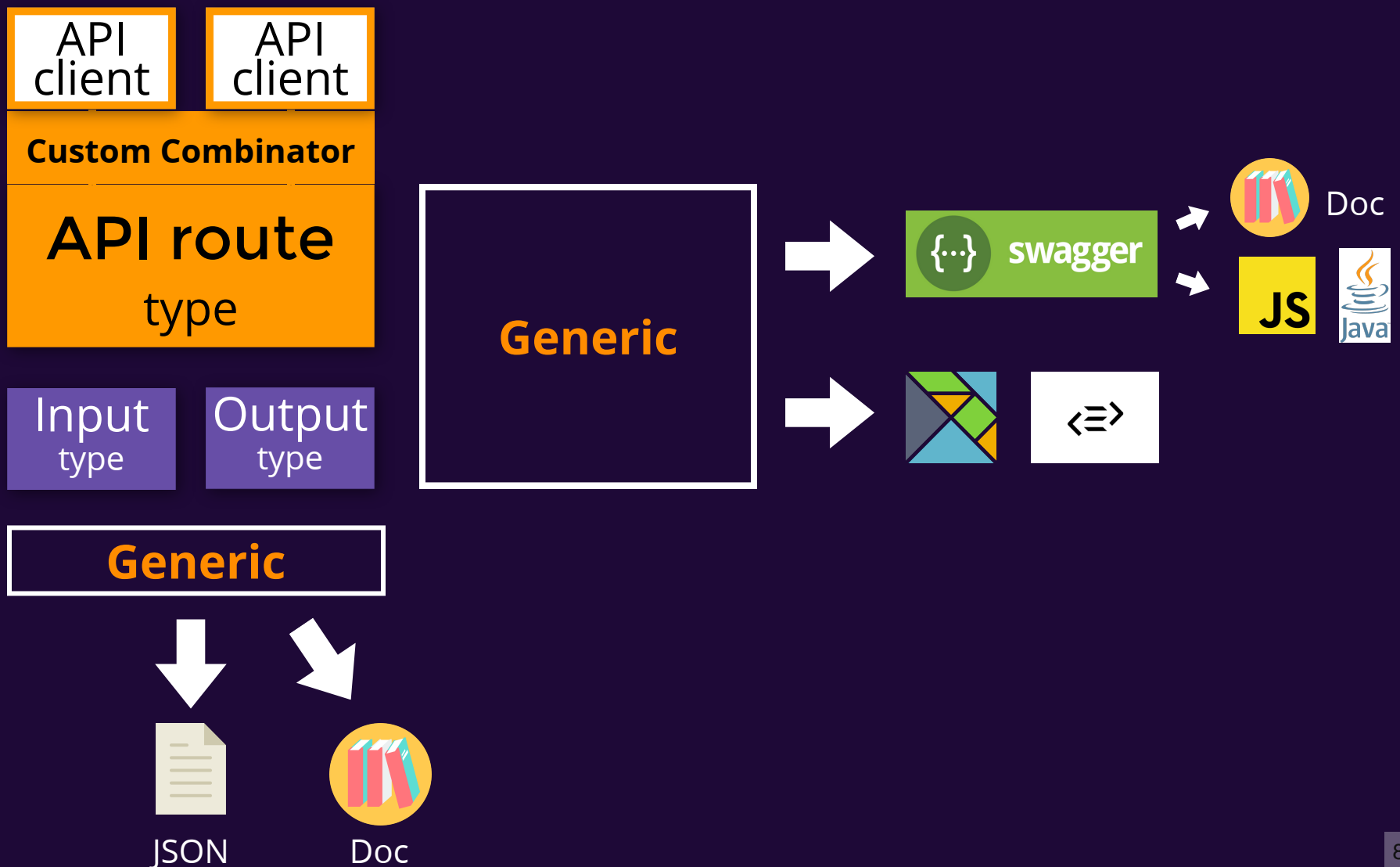
Monad m  **Monad n**

```
1 hoistServerWithContext :: Proxy (AuthUsers :> api)
2          -> Proxy (UsersContext ': context)
3          -> (forall x. m x -> n x)
4          -> ServerT (AuthUsers :> api) m
5          -> ServerT (AuthUsers :> api) n
```

Custom Combinator
DEMO

A Typesafe API

With Servant



HASKELL SONT JOLIES LES FEATURES DE L'API

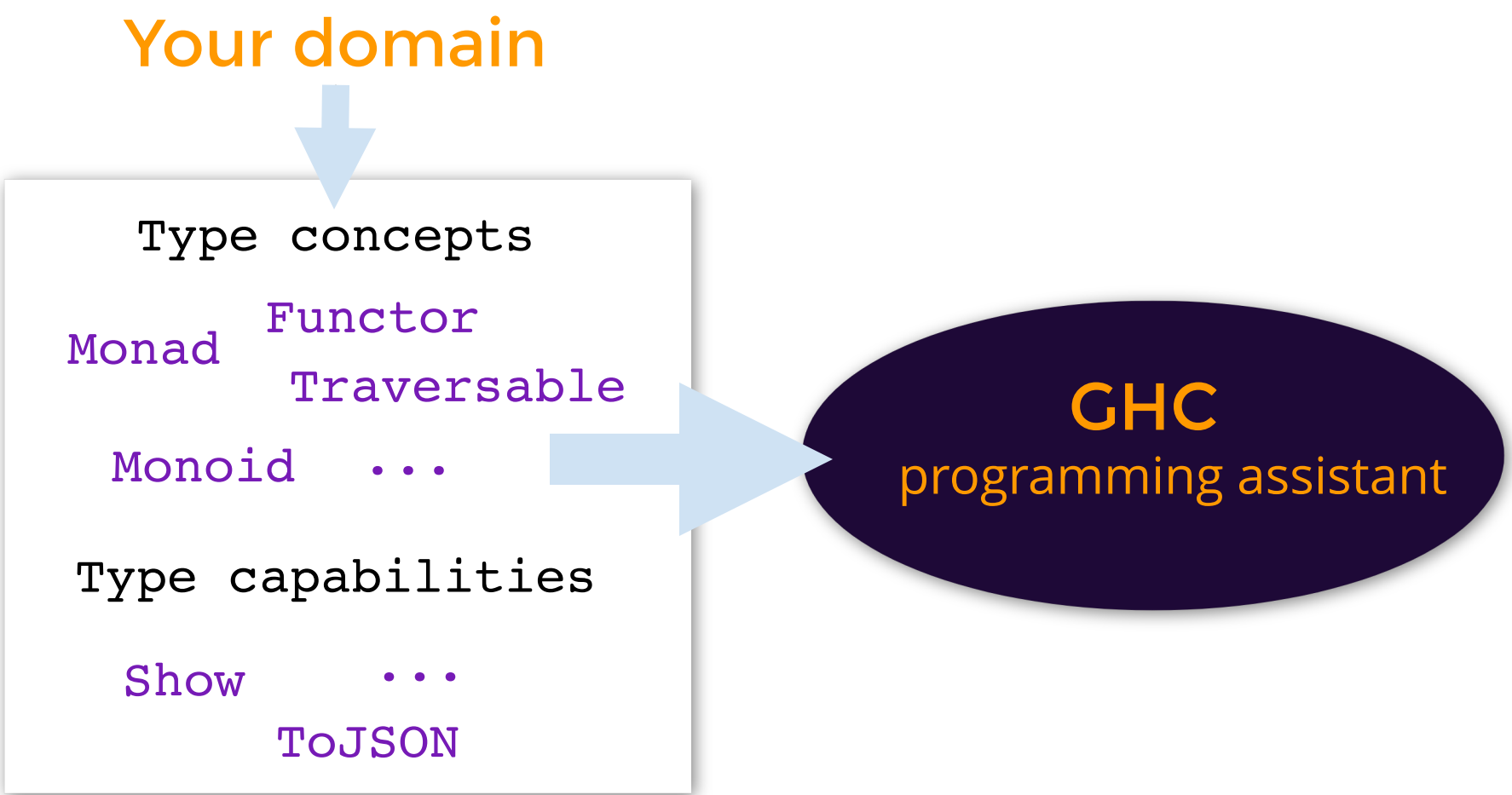
merci @ShirleyAlmCh ;-)

https://www.build-rh.fr/


Questions?

How Haskell works?

Implement your domain in the type system



Generic

Typeclass

```haskell
class Generic a where

  type Rep a :: * -> *

  from : a -> Rep a x

  to : Rep a x -> a
```

Generic

Implementing a generic

```haskell
class GEq a where
 geq :: a x -> a x -> Bool

instance GEq U1 where
 geq U1 U1 = True

instance GEq V1 where
 geq _ _ = True

instance Eq a => GEq (K1 _1 a) where
 geq (K1 x) (K1 y) = x == y

instance (GEq a, GEq b) => GEq (a :+: b) where
 geq (L1 x) (L1 y) = geq x y
 geq (R1 i) (R1 j) = geq i j
 geq _ _ = geq i j

instance (GEq a, GEq b) => GEq (a :*: b) where
 geq (x1 :*: y1) (x2 :*: y2) = geq x1 x2 && geq y1 y2

instance GEq a => GEq (M1 _x _y a) where
 geq (M1 x1) (M1 x2) = geq x1 x2
```

No param

Absurd

Constructors

Sum type

Product type

Metadatas

Generic

Implementing a generic

```haskell
class Eq a where
  eq :: a -> a -> Bool

  default eq :: (Generic a, Geq (Rep a)) => a -> a -> Bool
  eq a b = geq (from a) (from b)
```

```haskell
data Movie =
    Movie { title :: Title
          , genre :: [Genre]
          , year :: Year
          }
          deriving stock (Generic)
          deriving anyclass (Eq)
```

Combine many API

type

?