

# SAML. Exceptions.

Jaseem V V <[jaseem.vv@concur.com](mailto:jaseem.vv@concur.com)>

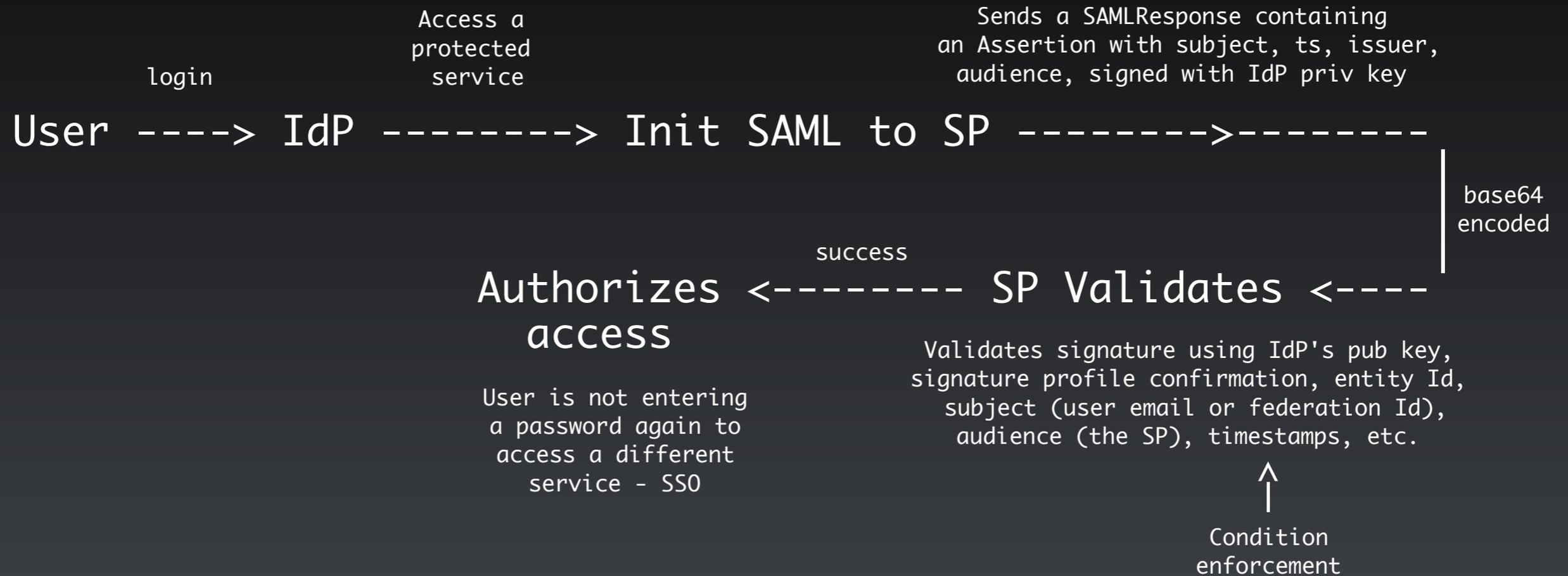
06.04.2017

# SAML SSO

- Security Assertion Markup Language – XML
- Request–response protocol
- Entities involved: IdP, SP, user
- IdP (Identity Provider): holds the credentials
- SP (Service Provider): provides the service
- Single Sign–On, from OASIS consortium
- Versions: 1.0, 1.1, 2.0, 2.1 (WIP)
- Flows: IdP, SP SAML

# IdP SAML

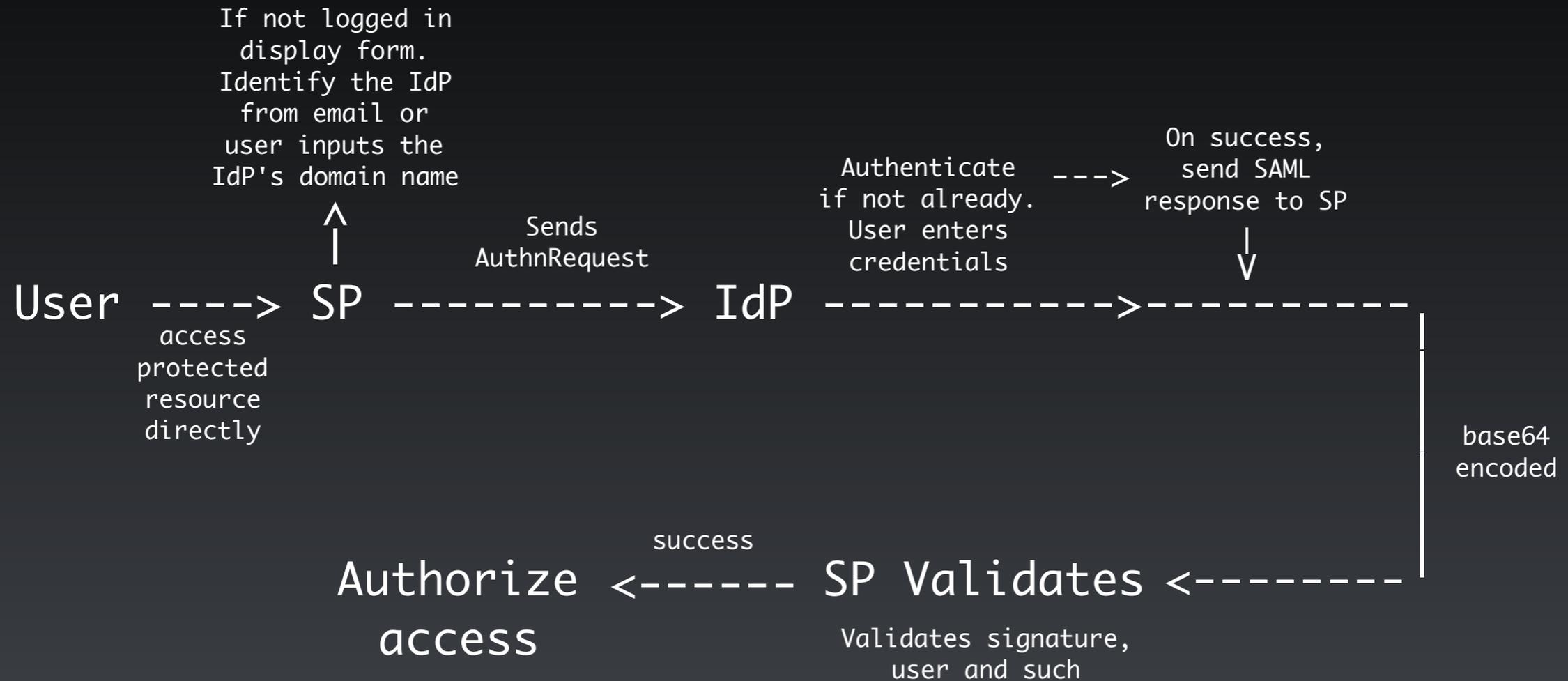
- Flow initiated from the IdP



- Eg: User signs into Boeing (IdP), access link to Concur portal for travel. IdP initiates SAML request to Concur and so on.

# SP SAML

- Flow initiated from SP



# Variant

- Also known as tagged union.

```
enum SzTag { short_t, ptr_t };
```

```
struct Data {  
    enum SzTag sz_tag;  
    union { /* 64 bits */  
        short short_val;  
        int *ptr_val;  
    }  
};
```

```
void process_data(struct Data *data) {  
    /* should we use short_val or ptr_val, how do we know? */  
    /* the tag comes handy */  
    if (data->sz_tag == short_t)  
        /* use data->short_val value */  
    } else if (data->sz_tag == ptr_t) {  
        /* use *(data->ptr_val) value */  
    }  
}
```

OS X LP64 size info

datatype	bytes	bits
short	2	16
pointer	8	64

- Sample program at GHE

## Variant in Clojure

- Use a vector with a tag and data `[:tag val]`
- Restrict to two elements for simplicity
- Use map for data if we need more values
- Not using a map as a variant – two lookups, for tag, value
- Goes in hand with pattern matching
  - destructure with `let`
  - cleaner using `match` from `core.match`

- Open variant => tag set is extensible, requires a default case
- Closed variant => well known tags, no default required
- Loop Variant (no theories)
  - Useful to decouple the recursion control logic
  - To control the number of recursions/iterations
  - Eg: workers, task distribution

```
[:stop return-val]
```

```
[:recur recur-val]
```

# Loop Variant Example

- Distributed sorting of a tree with child nodes
  - sort manager: controls recursion and work allocation
  - sort worker: sort algorithm impl returning a variant indicating whether to continue if child nodes exist

Pseudo code:

```
sort_worker() ->  
  % Sort top level nodes, check if any child node exist,  
  % if so use recur tag, else stop tag. Worker can be a remote node or local.
```

```
dist_sort(Node) ->  
  % Find a worker node based on some criteria, and send the Node to the sort_worker  
  % for processing. Does the distribution of jobs to workers & merging.
```

```
sort_manager(TaggedTuple) ->  
  {recur Node} -> [dist_sort(N) || N <- get_children(Node)]  
  {stop Node} -> Node
```

## Result Variant & core.match

- Commonly used in error cases which are not exceptions

```
[ :ok val ]  
[ :err msg ]  
[ :fatal { :cause "Node crashed", :req req-obj, :node-ip "192.168.10.13" } ]
```

- Use match for pattern matching
- Clean, powerful destructuring

```
(match (compute args) ;compute fn returns a variant  
  [ :ok result ] (merge result) ;success  
  [ :err msg ] (audit msg) ;do something with the error  
  [ :fatal { :cause cause, :node-ip ip } ] (sup [ :restart ip ])) ;notify supervisor
```

↑  
Selective destructure  
of map keys. This map data  
can have many keys but here  
our interest is in only two values.

- core.match macro is slower than stdlib

# Handle Exceptions

- try catch

```
(try
  ;code that can throw an exception
  (catch Exception excep
    ; ..))
```

- Ring middleware

```
(defn wrap-exceptions
  [handler]
  (fn [request]
    (try
      (handler request)
      (catch Exception excep
        (let [{[tag _] :cause} (ex-data excep)]
          (condp tag
            :err-signature (sig-err-resp)
            :err-user-inactive (usr-inactive-resp)
            ; ..
          )))
    )))
```

Fine grained exception responses without any clutter

Avoids having to use `try catch` everywhere. Middleware handles at route level. Catch all.

Simple destructuring. We can use `match` for complex cases. Matches exceptions thrown using `ex-info`.

Can have multiple `catch` for checked exceptions

- Above pattern goes in hand with throwing custom exceptions using `ex-info`

# Throw Custom Exceptions

- We can use `ex-info` to throw custom exceptions

```
(defn validate-issuer [issuer entity-id]
  (when-not (= issuer entity-id)
    (throw (ex-info "Issuer mismatch." {:cause [:err-issuer-mismatch issuer] }))))
```

Throw `ex-info` with `cause` as the variant with value having the wrong `issuer` obtained

→ Log it. Do further processing, by middleware

- Cases where it's not appropriate to handle the exception by middleware can have the function return a result variant instead of throwing exceptions

# Prelude to Monad

- Monad is a container typeclass.
- Comes from Category Theory. Heavily used in Haskell
- Used for composing (chaining), maintain state between them, side effects, especially in a pure functional language
- Container: holds values. Type of value give the type of the container. Eg: tree, graph, list etc. Means value is wrapped.

```
class Eq a where  
    (==) :: a -> a -> Bool
```

<code>==</code>	<code>::</code>	<code>a -&gt;</code>	<code>a -&gt;</code>	<code>Bool</code>
Operation provided by Class <code>Eq</code> which its instance <code>a</code> must define	has type	arg 1 with type <code>a</code>	arg 2 with same type <code>a</code>	and returns a boolean

```
instance Eq Integer where  
    x == y = x `integerEq` y
```



Type `Integer` implements `Eq` by providing `==` definition

# Category Theory

- Category is a collection of objects and arrows between objects
- Hask is the Haskell category where objects are types
- Arrows are known as morphisms
- Morphism means transformation
- Transformation under Hask category means functions
- Functor is a morphism from category to category
- Intuition: functor is a type class that can transform by mapping a function `fmap` over a collection of types and return a collection of new (lifted) types, where the same function can be applied again
- Functors in Haskell is an endofunctor since it maps within the same category, i.e., types

# Functor

```
class Functor f where
```

```
fmap :: (a -> b) -> f a -> f b
```

↑  
Takes a function  
of type `a` which  
returns type `b`

↑  
Returns a function  
that takes `f a` and  
return `f b` for some  
type constructor `f`

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

↑  
Maybe as a type

↑   ↑  
type constructors

```
class Functor Maybe where
```

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
```

↑  
Maybe as a  
functor  
typeclass  
definition

↑  
Takes a  
normal  
function

↑  
Returns a function  
of a lifted  
(more generic) type

```
instance Functor Maybe where
```

```
fmap _ Nothing = Nothing
```

```
fmap f (Just a) = Just (f a)
```

- Generalization: Endofunctors in the category of Hask returns a lifted type
- Functor in Hask should map to the same type constructor
- => No `Int` to lifted `String` type etc. `Int` to lifted `Int` only

# Functor Example

```
addThree :: Int -> Int
addThree = (+ 3) -- normal function
```

```
> addThree (Just 2) -- error
--- *** addThree takes an Int but you gave it a Maybe Int.
```

```
> fmap addThree (Just 2)
Just 5
```

Make it a functor. Since Just 2 is a Maybe constructor and Maybe is an instance of Functor, fmap knows how to apply the function to the wrapped value or context

```
> fmap addThree Nothing
Nothing
```

```
-- Infix style
> addThree <$> Just 2
Just 5
```

```
> addThree <$> Nothing
Nothing
```

We converted a normal function to a lifted function using a functor, and got free error handling

# Applicative Functor

- Simply put, an upgraded version of a functor
- Intuition: Takes a wrapped function(s) and a wrapped value(s), unwarp both, apply the each functions to each values and return a wrapped value of the result
- Eg using infix (sequence application ap):

```
Just addThree <*> Just 2 -- Just 5
```

↑                    ↑                    ↑  
Wrapped            Wrapped            Wrapped  
function            value            value as  
                    as arg            result

```
> [(*2), (+3)] <*> [1, 2, 3]  
[2, 4, 6, 4, 5, 6]
```

# Monad

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

↑            ↑            ↑            ↑

bind        wrapped value; instance of monad        function that takes a value and returns a monad        return a wrapped value (monad)

```
instance Monad Maybe where
  Nothing >>= func = Nothing
  Just val >>= func = func val
```

↑            ↑            ↑

wrapped value        function        apply function to the value

```
half :: Int -> Maybe
half x = if even x
        then Just (x `div` 2)
        else Nothing
```

the result of this function is again a wrapped value

```
> Just 20 >>= half >>= half >>= half
Nothing
```

↑

short circuits.  
5 not even

Can compose as many functions as we want. Here we use the same function `half`, it can be any that matches the type.

We got exception handling. Only the final value need to be unwrapped if success.

- Maybe is a functor, applicative, monad and many other things



- Monads gets composed
- Except monad builds on top of Either monad and such
- Monad is an abstraction and can be used in any language
- It's more elegant in a pure functional language with an intelligent type system (HM)

- Snippets from oauth2

```
(defmonad maybe-err-m
  ;;
  ;; runs computations in sequence. Returns current state of computation
  ;; if any step fails
  ;;
  [m-result (fn [x] x)

  m-bind (fn [mx f]
           (if (failed? mx) mx (f mx)))

  ])
```

```
(defmacro ==>
  "Given map m & arity-1 fns, this macro runs the fns in order, supplying the map m
  to the first fn. All fns must accept a map. The fns are chained with m-chain (like
  the -> macro) under the maybe-err-m monad.

  if the monad short-circuits the computation, this macro will run the error-function
  stored as (:efn m)"
  [m & vfns]
  (let [payload (last vfns)
        fns (vec (drop-last vfns))]
    `(let [z# ((with-monad maybe-err-m
                (m-chain ~fns)) ~m)]
        (if (failed? z#)
            (:efn z#) z#
            (~payload z#))))))
```

# Links

- clojure/conj variants errata + video  
<http://jneen.net/posts/2014-11-23-clojure-conj-variants-errata>
- variant sample  
<http://github.concur.com/jaseemv/variant>

Thank You