# Show me your Authorizations, and I will show mine

pertti.kellomaki@solita.fi

# Context of the Talk: Taitorakennerekisteri

"Only members of the owner organization with role 'builder' may modify a bridge and its associated documents"

# Problems...

- Complex authorization conditions:
  - Roles, ownership, authorship, contractual relations, lifecycle stages, visibility constraints, ...
- Authorization code has become complex, and consequently difficult to modify, test, and validate
- No automatic way to extract authorization conditions from code
  => documentation of authorizations maintained manually

# A Solution: Simple DSL

- **Variables** hold data used in authorization
  - `:USERID`, `:BRIDGEID`, `:DOCUMENTID`, ...
- **Operators** express relationships between variables
  - `and`, `or`, `not`, `=`, ...
- **Syntax** looks like code, but is in reality (Clojure) data
  - `(and (= :BRIDGES-OWNER :USERS-ORGANIZATION) (= :USERS-ROLE :builder))`

- Easy to do in Clojure, but the approach is not Clojure specific

# Benefits

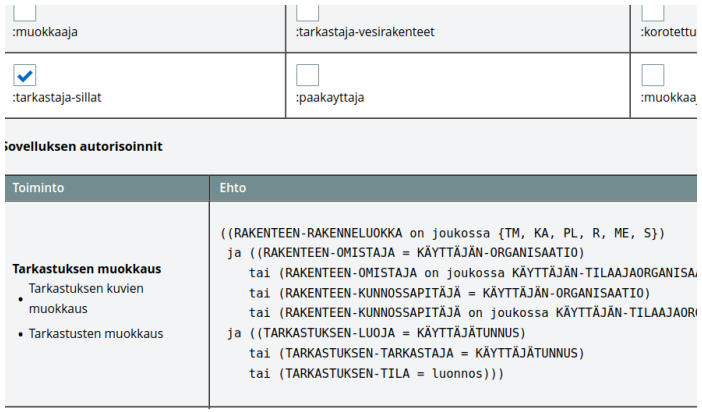
- Single source of truth
  - Authorization of requests => evaluate DSL expressions
  - Documentation => traverse DSL expressions to produce customer readable "natural" language descriptions
- Bonus
  - Pose questions using partial evaluation, e.g. "What can a user with role `builder` do?"
    - Start with authorization conditions in DSL
    - Partially evaluate them assuming values for DSL variables

# Example: Taitorakennerekisteri

Full authorization condition

| Toiminto | Ehto |
| --- | --- |
| **Tarkastuksen muokkaus**<br>• Tarkastuksen kuvien muokkaus<br>• Tarkastusten muokkaus | ((paakayttaja on joukossa KÄYTTÄJÄN-ROOLIT)<br>tai (laaja-muokkaaja on joukossa KÄYTTÄJÄN-ROOLIT)<br>tai (((paakayttaja on joukossa KÄYTTÄJÄN-ROOLIT)<br>tai (tarkastaja on joukossa KÄYTTÄJÄN-ROOLIT)<br>tai (tarkastaja-sillat on joukossa KÄYTTÄJÄN-RO<br>tai (tarkastaja-tunnelit on joukossa KÄYTTÄJÄN-<br>tai (tarkastaja-vesirakenteet on joukossa KÄYTT<br>ja (((tarkastaja-sillat on joukossa KÄYTTÄJÄN-ROOL<br>ja (RAKENTEEN-RAKENNELUOKKA on joukossa {TU<br>tai ((tarkastaja-tunnelit on joukossa KÄYTTÄJÄN<br>ja (RAKENTEEN-RAKENNELUOKKA on joukossa {KA<br>tai ((tarkastaja-vesirakenteet on joukossa KÄYT<br>ja (RAKENTEEN-RAKENNELUOKKA on joukossa {TU<br>ja ((RAKENTEEN-OMISTAJA = KÄYTTÄJÄN-ORGANISAATIO)<br>tai (RAKENTEEN-OMISTAJA on joukossa KÄYTTÄJÄN-T<br>tai (RAKENTEEN-KUNNOSSAPITÄJÄ = KÄYTTÄJÄN-ORGAN<br>tai (RAKENTEEN-KUNNOSSAPITÄJÄ on joukossa KÄYTT<br>ja ((TARKASTUKSEN-LUOJA = KÄYTTÄJÄTUNNUS)<br>tai (TARKASTUKSEN-TARKASTAJA = KÄYTTÄJÄTUNNUS)<br>tai (TARKASTUKSEN-TILA = luonnos)))) |


Partial evaluation:
User is admin user




Partial evaluation:
User is bridge inspector

# Where do DSL Variables Come from?

Request variables

```
{:document
  {:id         11
   :bridgeid 1
   :contents "Tower Bridge updated blueprints"}}
```

PUT "/documents", user "bob"

server

db

| Document.id | Document.bridgeid |
|---|---|
| 11 | 1 |

| Bridge.id | Bridge.owner |
|---|---|
| 1 | Acme Inc. |

Derived variables

| User.id | User.organization | User.role |
|---|---|---|
| bob | Acme Inc. | builder |

Only members of the owner organization with role 'builder' may modify a bridge and its associated documents

# DSL: Request Variables

```
(def routes
  {"/documents"
    :PUT {:parameters
            {:document Document}
          :handler ...
  ...
```

```
(defschema Document
  (->
   {:id         DocumentId
    :bridgeid   BridgeId
    :contents   schema/Str}
   (auth-data/attach
        {:DOCUMENTID :id
         :BRIDGEID   :bridgeid})))
```

Payload

Instructions for extracting DSL variables

```
(auth-data/extract
  {:id 11 :bridgeid 1 :contents "Tower Bridge updated blueprints"}
  Document)
 =>
  {:DOCUMENTID {:operator :constant, :value 11},
   :BRIDGEID   {:operator :constant, :value 1}}
```

Extracting DSL variables when authorizing a request

# DSL: Derived Variables

```
(def variables
  {:BRIDGES-OWNER       [db/get-bridges-owner :BRIDGEID]
   :DOCUMENTS-BRIDGEID [db/get-documents-bridgeid :DOCUMENTID]}
 ...
```

- The value of a derived variable is obtained by calling the deriving function with the result of evaluating the variable it depends on
- A derived variable can depend on another derived variable, resulting in traversal of the dependency chain

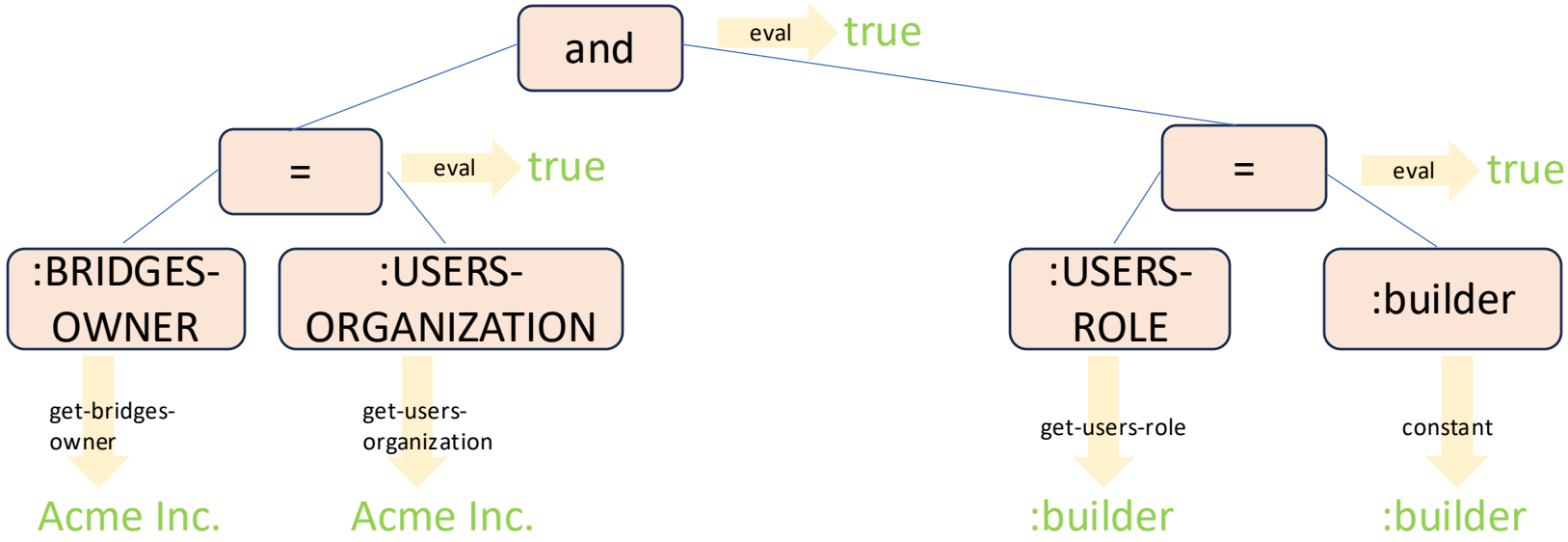
# Evaluation

Initial variables

```
:USERID "bob"
:BRIDGEID 1
```

Expression

```
(and (= :BRIDGES-OWNER :USERS-ORGANIZATION) (= :USERS-ROLE :builder))
```




| Document .id | Document .bridgeid |
|---|---|
| 11 | 1 |

| Bridge .id | Bridge .owner |
|---|---|
| 1 | Acme Inc. |

| User .id | User .organization | User .role |
|---|---|---|
| bob | Acme Inc. | builder |

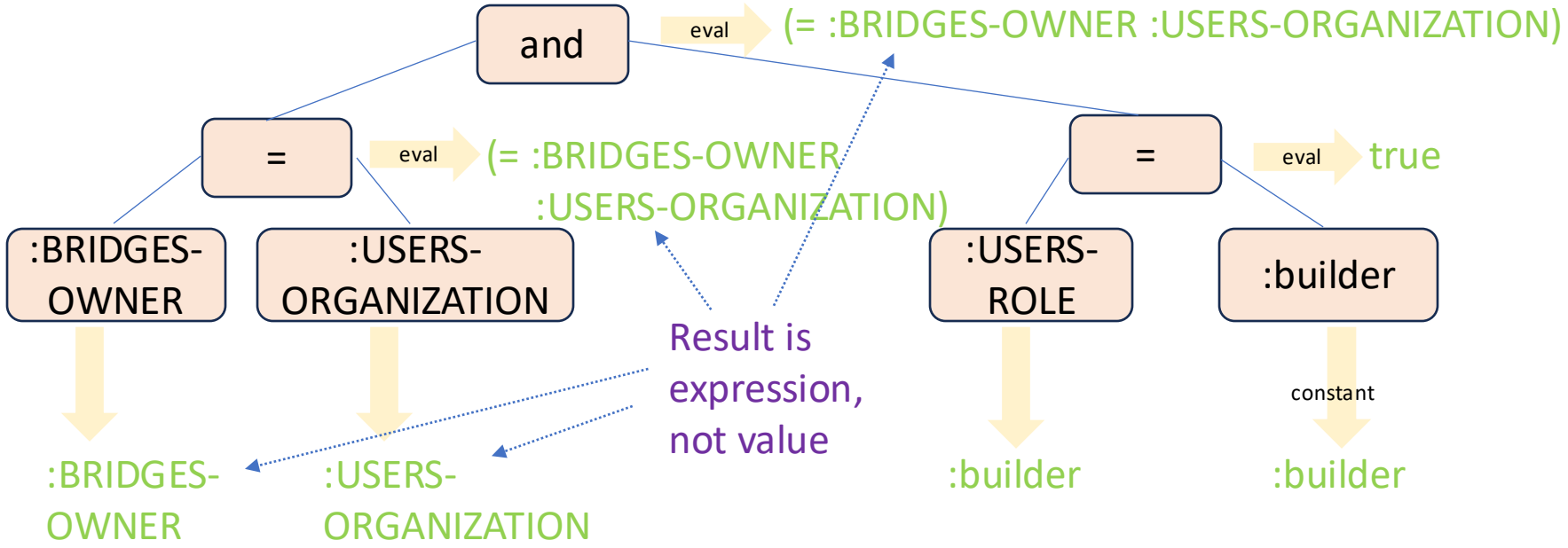
# Partial Evaluation (Expression Simplification)

Initial variables

:USERS-ROLE :builder

Expression

```
(and (= :BRIDGES-OWNER :USERS-ORGANIZATION) (= :USERS-ROLE :builder))
```




| Document .id | Document .bridgeid |
|---|---|
| 11 | 1 |

| Bridge .id | Bridge .owner |
|---|---|
| 1 | Acme Inc. |

| User .id | User .organization | User .role |
|---|---|---|
| bob | Acme Inc. | builder |

# Documenting with Partial Evaluation

What is the authorization condition for modifying bridges?

```
(explain bridge-modification-auth)
 => "(:BRIDGES-OWNER = :USERS-ORGANIZATION) and (:USERS-ROLE = :builder),
     where :BRIDGEID is determined by [:BRIDGEID, :DOCUMENTS-BRIDGEID]"
```

When can a builder modify a bridge?

```
(explain
  (eval-expr {:USERS-ROLE (create-constant :builder)}
             bridge-modification-auth))
 => ":BRIDGES-OWNER = :USERS-ORGANIZATION,
     where :BRIDGEID is determined by [:BRIDGEID, :DOCUMENTS-BRIDGEID]"
```

When can a mere mortal modify a bridge?

```
(explain
  (eval-expr {:USERS-ROLE (create-constant :mere-mortal)}
             bridge-modification-auth))
 => "false"
```

# Putting it all Together

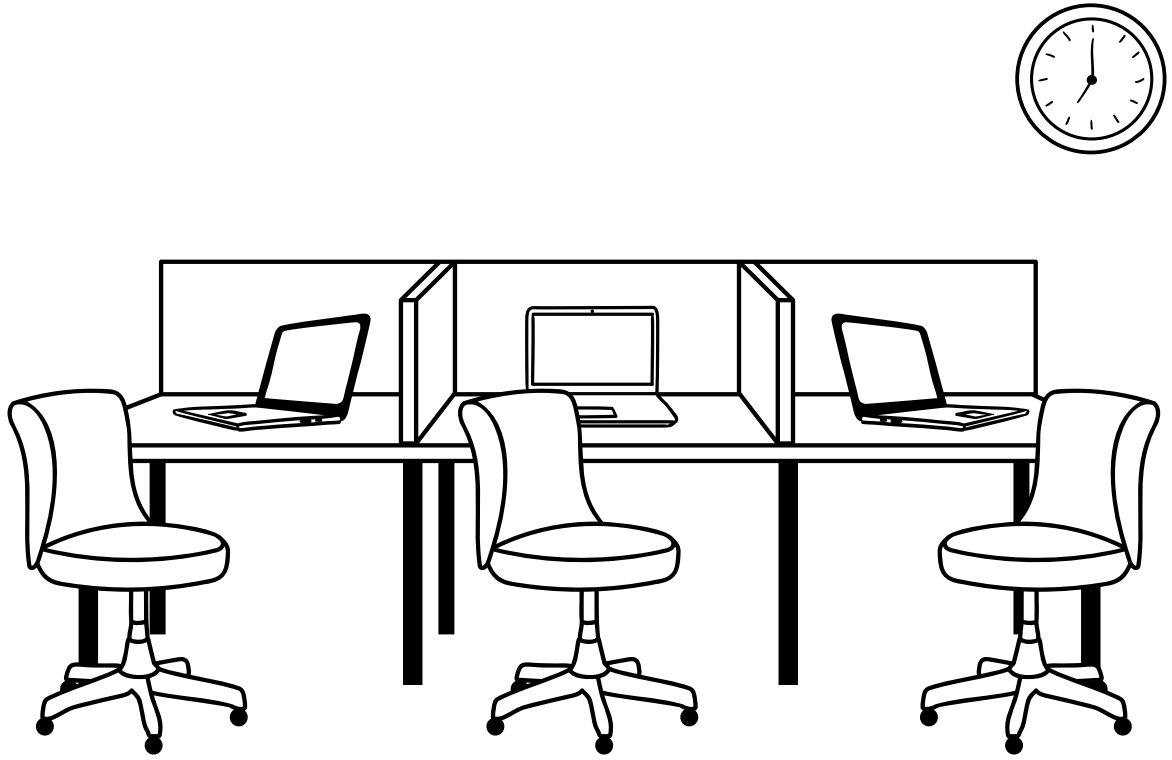
```
(defschema Document
  (-> {:id         DocumentId
       :bridgeid BridgeId
       :contents  schema/Str}
      (auth-data/attach
        {:DOCUMENTID :id
         :BRIDGEID   :bridgeid})))
```

```
(def routes
  {"/documents"
   {:PUT  {:parameters {:document Document}
           :authorization bridge-modification-auth
           :handler    handlers/modify-document}
    :POST {:parameters {:document Document}
           :authorization bridge-modification-auth
           :handler    handlers/create-document}
    ...
```

```
(def bridge-modification-auth
 (parse
   '(unique [:BRIDGEID [:BRIDGEID
                        :DOCUMENTS-BRIDGEID]]
            (and (= :BRIDGES-OWNER
                    :USERS-ORGANIZATION)
                 (= :USERS-ROLE
                    :builder)))))
```

```
(explain bridge-modification-auth)


=> "(:BRIDGES-OWNER = :USERS-ORGANIZATION)
     and (:USERS-ROLE = :builder),
    where :BRIDGEID is determined by
     [:BRIDGEID, :DOCUMENTS-BRIDGEID]"
```

# That's it. Now please return to your cubicle.



https://github.com/solita-perttikellomaki/devday-authorization.git

# Authorization in the Routes

```
(def public-access-auth (parse true))
(def bridge-creation-auth (parse '(= :USERS-ROLE :builder)))
(def bridge-modification-auth
  (parse '(unique [:BRIDGEID [:BRIDGEID :DOCUMENTS-BRIDGEID]]
                            (and (= :BRIDGES-OWNER :USERS-ORGANIZATION) (= :USERS-ROLE :builder)))))

(def routes
  {"/bridges"
   {:GET  {:parameters {:id BridgeId}
           :authorization public-access-auth
           :handler    handlers/get-bridge}
    :PUT  {:parameters {:bridge Bridge}
           :authorization bridge-modification-auth
           :handler    handlers/modify-bridge}
    :POST {:parameters {:bridge Bridge}
           :authorization bridge-creation-auth
           :handler    handlers/create-bridge}}
   "/documents"
   {:GET  {:parameters {:id DocumentId}
           :authorization public-access-auth
           :handler    handlers/get-document}
    :PUT  {:parameters {:document Document}
           :authorization bridge-modification-auth
           :handler    handlers/modify-document}
    :POST {:parameters {:document Document}
           :authorization bridge-modification-auth
           :handler    handlers/create-document}}})
```

# Authorization in the Server

```
(defn simple-server [routes]
 (fn serve [address method parameters]
   (let [handler (get-in routes [address method :handler])]




    (handler parameters))))
```

```
(defn authorizing-server [routes]
  (fn serve [address method userid parameters]
    (let [handler           (get-in routes [address method :handler])
          auth-condition    (get-in routes [address method :authorization])
          route-parameters  (get-in routes [address method :parameters])
          request-variables (->> route-parameters
                                 (map (fn [[param schema]]
                                        (extract (param parameters) schema)))
                                 (apply merge))]
      (let [auth-result
            (eval-expr
              (assoc request-variables
                     :USERID (create-constant userid))
              auth-condition)]

        (if (constant-true? auth-result)
          (handler parameters)
          {:error "Access denied"})))))
```

Disclaimer: in a real implementation you need to consider what to do if request variables yield conflicting values for request variables.

# Authorization Language: Operators

```
(def bridge-modification-auth
  (parse
    '(unique [:BRIDGEID [:BRIDGEID :DOCUMENTS-BRIDGEID]]
             (and (= :BRIDGES-OWNER :USERS-ORGANIZATION)
                  (= :USERS-ROLE :builder)))))
```

```
(explain bridge-modification-auth)
 => "(:BRIDGES-OWNER = :USERS-ORGANIZATION) and (:USERS-ROLE = :builder),
     where :BRIDGEID is determined by [:BRIDGEID, :DOCUMENTS-BRIDGEID]"
```

# Evaluation

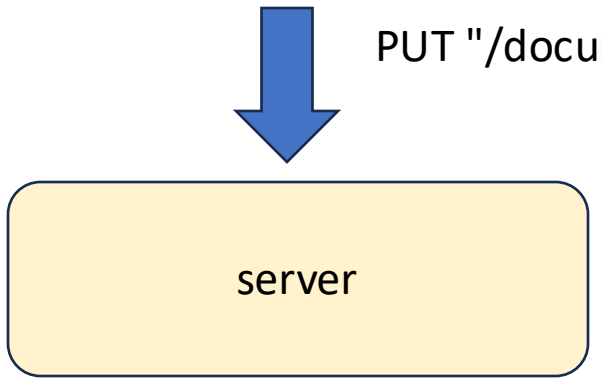
```
(defn eval-expr [env expr]
  (cond (constant? expr)
        expr

        (variable? expr)
        ((get variable-semantics (:name expr)) env expr)

        (= (:operator expr) :unique)
        (eval-unique env expr)

        :else
        ((get operator-semantics (:operator expr))
         (map (fn [ex] (eval-expr env ex))
              (:operands expr)))))
```

Disclaimer: in a real implementation you probably want to thread env through traversal of the expression and accumulate values of derived variables in order compute them just once.

# The World of Taitorakennerekisteri

# Taitorakennerekisteri at Work

```
{:document
  {:id       11
   :bridgeid 1
   :contents "Tower Bridge updated blueprints"}}
```




```
(def routes
  {"/bridges"
   {:GET  {:parameters {:id BridgeId}
           :handler    handlers/get-bridge}
    :PUT  {:parameters {:bridge Bridge}
           :handler    handlers/modify-bridge}
    :POST {:parameters {:bridge Bridge}
           :handler    handlers/create-bridge}}
   "/documents"
   {:GET  {:parameters {:id DocumentId}
           :handler    handlers/get-document}
    :PUT  {:parameters {:document Document}
           :handler    handlers/modify-document}
    :POST {:parameters {:document Document}
           :handler    handlers/create-document}}})
```