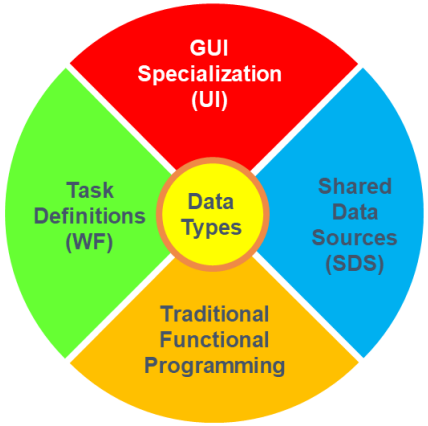
Task Oriented Programming

Sven-Bodo Scholz, **Peter Achten**

Advanced Programming

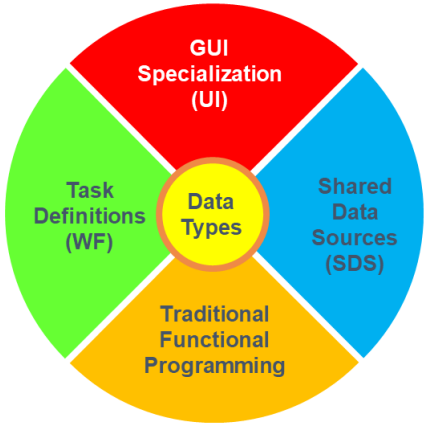
part 2/2

Recap lecture 1/2



- Task Oriented Programming
- DSL for applications that coordinate people and systems
- Capture domain and relations with data types and pure functions
  - `derive class iTask T`
- User interaction: editor tasks ((`enter`/`view`/`update`)`Information`)
  - editor tasks never have a stable task value
- Transform a task value (`@` and `@!`)
- Stable task value (`return`)
- Sequential task composition (`>>*`, `>>-`, `>>?`, `>?|`, …)
- Parallel task composition (`parallel`, `allTasks`, `anyTask`, `-&&-`, `-||-`, `-||`, `||-`, …)

What this lecture is about



- Shared Data Sources: abstraction over (persistent) data
- Distributed Systems: workers and how to assign work to them
- Guest lecture: a real-world industrial strength TOP application

Shared Data Sources

Shared Data Sources - TOP

- There are many sources of information:
  - shared memory, files, data bases, cloud, time, sensors, human resources, …
- Instead of programming them on an individual basis, we aim for a uniform abstraction and interface: Shared Data Source (SDS)
- SDSs should interact with tasks
- SDSs should be compositional

Shared Data Sources - iTask

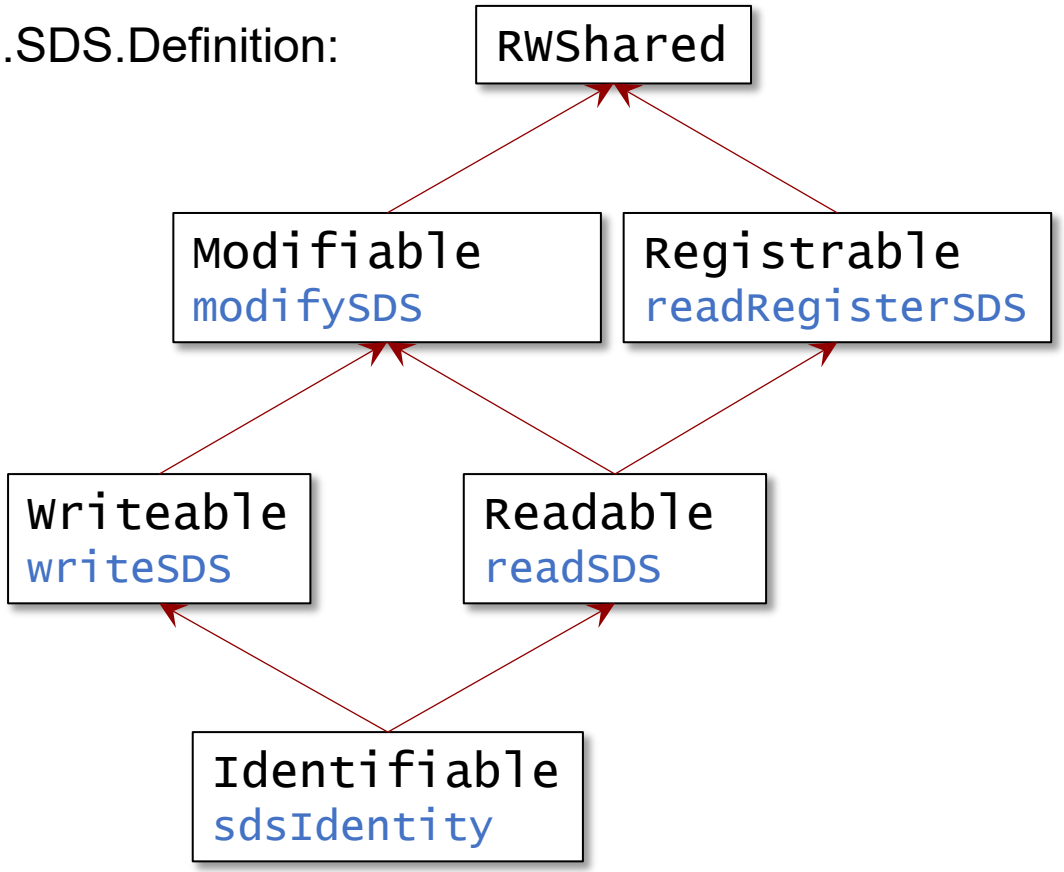
```
:: SDSLens p r w
```

- Reading `r` and writing `w` can be of different types
- Publish subscribe system for efficient updates
- Parametric lenses to focus on parts of the SDS (`p`)
- The types are complicated, use is much less so

```
:: SimpleSDSLens a :== SDSLens () a a
```

Shared Data Sources - iTask

iTasks.SDS.Definition:



iTasks.Internal.SDS:

```
instance Identifiable SDSLens
instance Readable     SDSLens
instance Writeable    SDSLens
instance Modifiable   SDSLens
instance Registrable  SDSLens
```

Example

```
currentUTCTime :: SDSLens () Time ()
currentUTCDate :: SDSLens () Date ()
```

iTasks.SDS.Sources.System

```
:: Date = { year :: !Int
          , mon  :: !Int
          , day  :: !Int
          }
:: Time = { hour :: !Int
          , min  :: !Int
          , sec  :: !Int
          }
```

iTasks.Extensions.DateTime

Example

```
currentUTCTime :: SDSLens () Time ()
currentUTCDate :: SDSLens () Date ()

viewSharedInformation
            :: ![ViewOption r]
               !(sds () r w)
            -> Task r | iTask r & TC w & RWShared sds

viewInformation
            :: ![ViewOption m]
               !m
            -> Task m | iTask m
```

when sds changes,
(viewSharedInformation … sds)
changes along

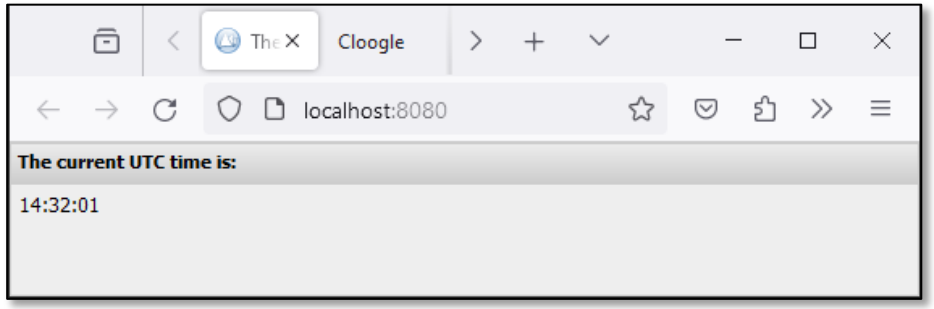
Example

```
currentUTCTime :: SDSLens () Time ()
currentUTCDate :: SDSLens () Date ()

viewSharedInformation
            :: ![ViewOption r]
               !(sds () r w)
            -> Task r | iTask r & TC w & RWShared sds

viewCurrentUTCTime :: Task Time
viewCurrentUTCTime
  = viewSharedInformation [] currentUTCTime
    <<@ Title "The current UTC time is:"
```

viewCurrentUTCTime
changes every second

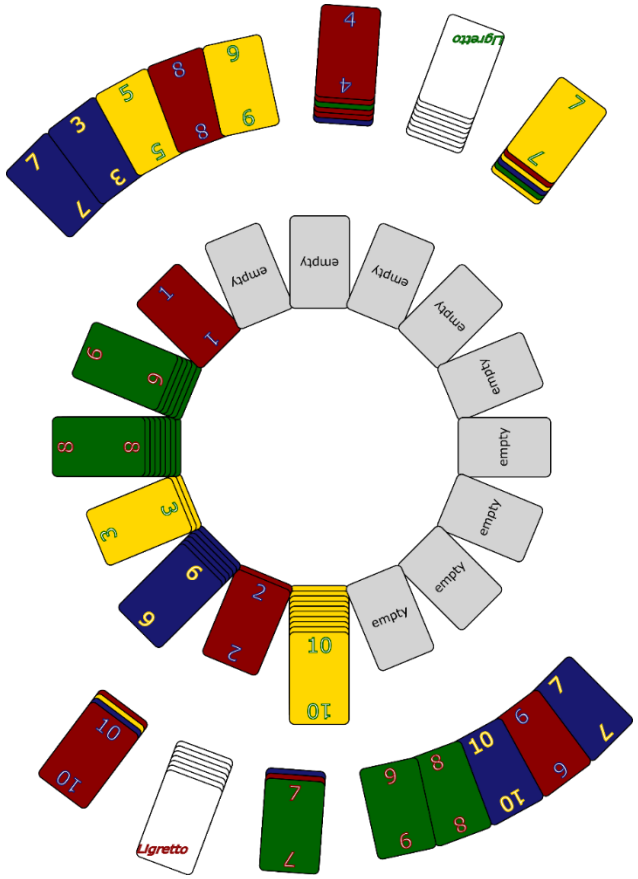


Example

```
currentUTCTime :: SDSLens () Time ()
currentUTCDate :: SDSLens () Date ()

updateSharedInformation
            :: ![UpdateSharedOption r w]
               !(sds () r w)
            -> Task r | iTask r & iTask w & RWShared sds

updateInformation
            :: ![UpdateOption m]
               !m
            -> Task m | iTask m
```



Combine tasks and SDSs

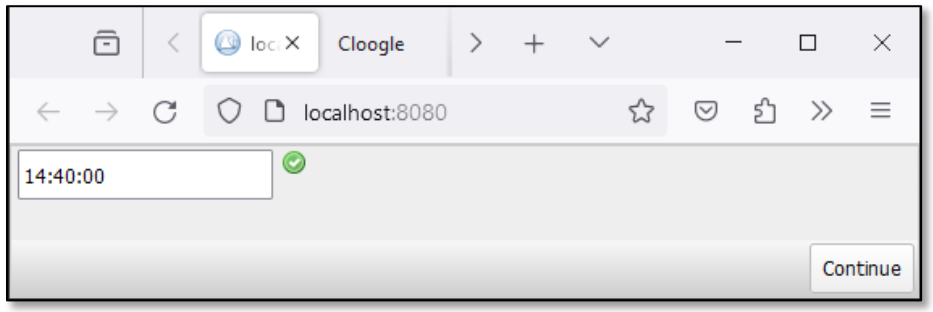
```
get   ::              !(sds () a w) -> Task a    | TC a & TC w & Readable  sds
set   :: !a           !(sds () r a) -> Task a    | TC a & TC r & Writeable sds
upd   :: !(r -> w)    !(sds () r w) -> Task w    | TC r & TC w & RWShared  sds
watch ::              !(sds () r w) -> Task r    | TC r & TC w
                                                & Registrable,  Readable  sds
wait  :: (r -> Bool) !(sds () r w) -> Task r | TC w & iTask r & RWShared  sds
```

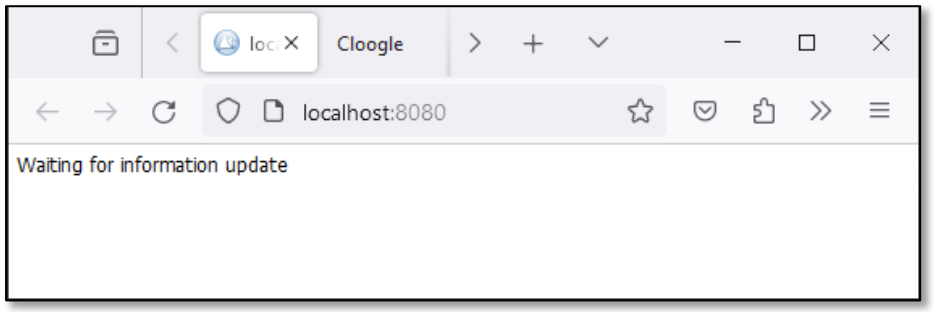
- (`get sds`) reads `sds` once, and returns a stable value
- (`set x sds`) write `x` to `sds`, and return `x` as stable value
- (`upd f sds`) atomic update of `sds` using `f` on current content of `sds`
- (`watch sds`) lifts `sds` to a task with unstable task values
- (`wait p sds`) waits until `p` holds for current content of `sds`, and returns a stable value

Example

```
reminders :: Time -> Task ()
reminders t
 =              updateInformation [] t <<@ Title "Set next reminder"
   >>? \then = wait ((<=) then) currentUTCTime
   >>- \done = reminders done

Start :: *World -> *World
Start world = doTasks (get currentUTCTime >>- reminders) world
```





Persistent SDSs

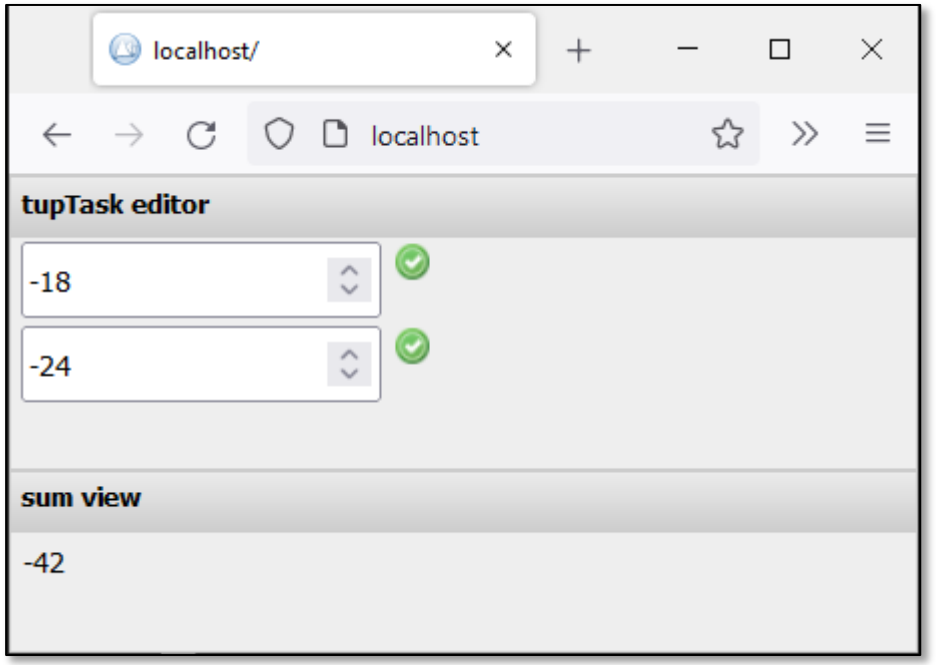
```
sharedStore :: !String !a
            -> SimpleSDSLens a | JSONEncode{|*|}
                               , JSONDecode{|*|}, TC a
```

- SDS accessible anywhere in application, using the name (first argument)
- persistent until next compilation (to prevent type and serialization issues)

Example

```
mySDS = sharedStore "mySDS" (1,2)

tupTask
 = ( updateSharedInformation [] mySDS
     <<@ Title "tupTask editor"
   ) -||
   ( viewSharedInformation [ViewAs (uncurry (+))] mySDS
     <<@ Title "sum view"
   )
```



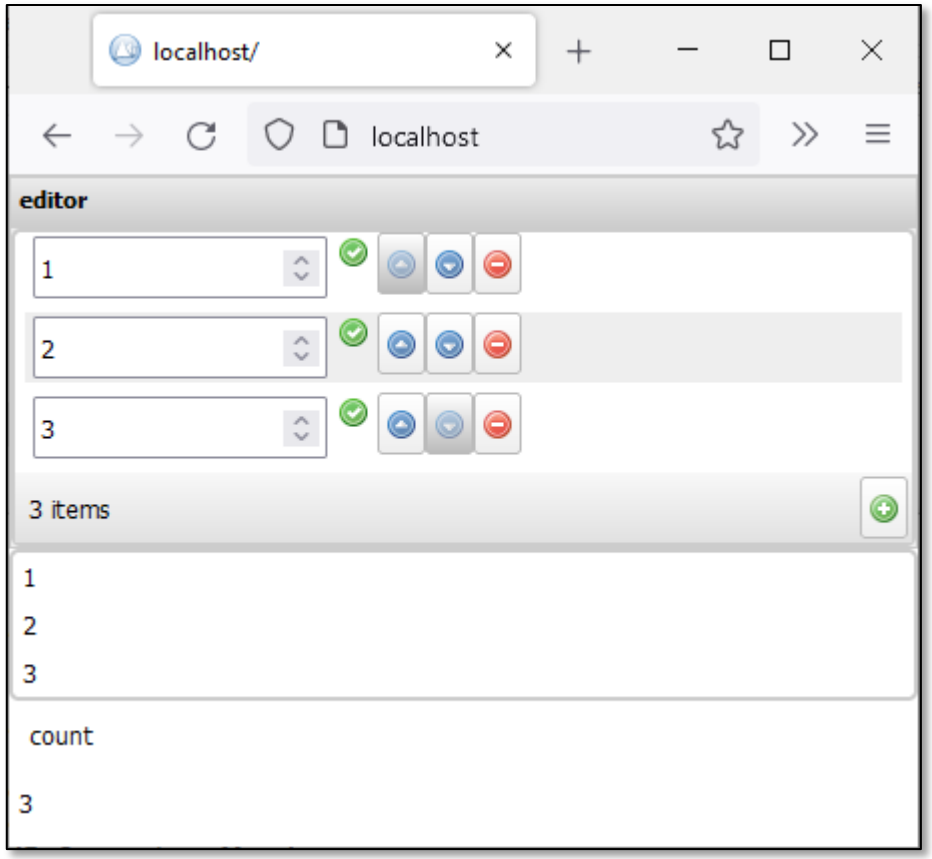
Locally scoped SDSs

```
withShared :: !b !((SimpleSDSLens b) -> Task a)
           -> Task a | iTask a & iTask b
```

- local SDS created, accessible only within task abstraction

Example

```
enterSharedList :: Task [Int]
enterSharedList
 = withShared [] (\sds
     = ( Title "editor" @>>
         updateSharedInformation [] sds
       )    -||
         viewSharedInformation [] sds
           -||
       ( viewSharedInformation [ViewAs length] sds
         <<@ Hint "count"
       )
   )
```



SDS combinator functions

```
(>*<) infixl 6 :: !(sds1 p rx wx) !(sds2 p ry wy) -> SDSParallel p (rx,ry) (wx,wy)
   | TC, gHash{|*|} p & TC rx & TC ry & TC wx & TC wy & RWShared sds1 & RWShared sds2
(>*|) infixl 6 :: !(sds1 p rx wx) !(sds2 p ry wy) -> SDSParallel p (rx,ry) wx
   | TC, gHash{|*|} p & TC rx & TC ry & TC wx & TC wy & RWShared sds1 & Registrable sds2
(|*<) infixl 6 :: !(sds1 p rx wx) !(sds2 p ry wy) -> SDSParallel p (rx,ry) wy
   | TC, gHash{|*|} p & TC rx & TC ry & TC wx & TC wy & Registrable sds1 & RWShared sds2
(|*|) infixl 6 :: !(sds1 p rx wx) !(sds2 p ry wy) -> SDSParallel p (rx,ry) ()
   | TC, gHash{|*|} p & TC rx & TC ry & TC wx & TC wy & Registrable sds1 & Registrable sds2
```

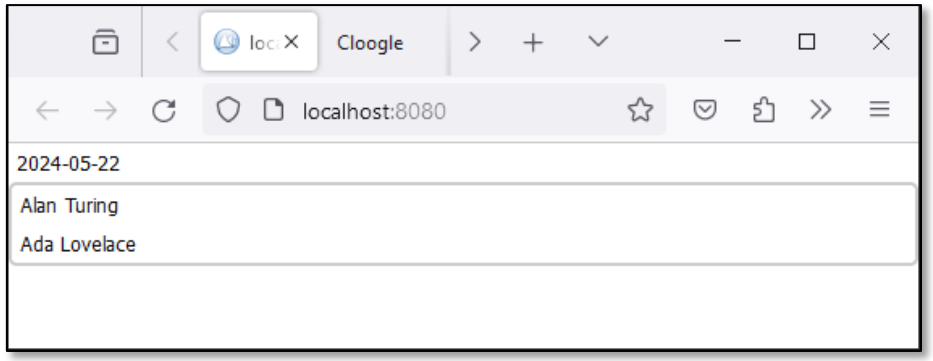
- `SDSParallel` is an instance of `RWShared`
- When sds1 or sds2 changes, their combination changes along
- Writing to the combination of sds1 and sds2 updates the proper sds (both, 1, 2, none)

Example

```
personsSDS :: SimpleSDSLens [Person]
personsSDS
 = sharedStore "hall of fame" []

name :: Person -> String
name {Person | name} = name

dailyPersonsOverview :: Task [Person]
dailyPersonsOverview
 = viewSharedInformation
      [ViewAs (\(today,persons) = (today,map name persons))]
      (currentUTCDate |*| personsSDS) @ snd
```



`dailyPersonsOverview` refreshes at start of day

[1] `:: Person` as defined in lecture 1/2

Shared Data Sources – Wrap Up

- SDSs provide uniform, typed, interface to tasks
- SDSs are compositional
- The iTask system uses SDS for internal administration

Distributed Systems

Distributed systems: user administration

iTasks.Extensions.Admin.UserAdmin:

```
:: User = SystemUser
        | AnonymousUser     !String
        | AuthenticatedUser !UserId ![Role] !(?DisplayName)
:: StoredUserAccount = { credentials        :: !StoredCredentials
                       , displayName        :: !?DisplayName
                       , roles              :: ![Role]
                       , token              :: !?StoredCredentials }
:: StoredCredentials = { username           :: !Username
                       , saltedPasswordHash :: !String
                       , salt               :: !String }
derive class iTask StoredUserAccount, StoredCredentials

userAccounts  ::          SDSLens () [StoredUserAccount] [StoredUserAccount]
users         ::          SDSLens () [User] ()
usersWithRole :: !Role -> SDSLens () [User] ()
```

Distributed systems: workflows

iTasks.Extensions.Admin.WorkflowAdmin:

```
:: WorkflowCollection
 = { name            :: !String
   , loginMessage    :: !?HtmlTag
   , welcomeMessage  :: !?HtmlTag
   , allowGuests     :: !Bool
   , workflows       :: ![Workflow]
   }
instance Startable WorkflowCollection

workflow :: String String w -> Workflow | toWorkflow w

instance toWorkflow      (Task a) | iTask a
instance toWorkflow (a -> Task b) | iTask a & iTask b
```

so it can be used in `doTasks`

embed in `WorkflowCollection`

Distributed systems: assigning tasks

- Assigning task to a user:

```
(@:) infix 3 :: !worker !(Task a) -> Task a | iTask a
                                            & toUserConstraint worker

:: UserConstraint
   = AnyUser
   | UserWithId !UserId
   | UserWithRole !Role
```

- `(@:)` is an application of `parallel` (lecture 1/2):
  - one embedded task for the current user is created
  - one detached task for the assigned user is created
  - local SDS is used for embedded task to check when detached task produces a value

Example

```
delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who    = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"


Start :: *World -> *World
Start world
 = doTasks { WorkflowCollection
           | name           = "Experiments2"
           , workflows      = [workflow "Delegate" "Palindrome" (delegate palindrome)]
           , loginMessage   = ?Just message
           , welcomeMessage = ?None
           , allowGuests    = False } world
where message = DivTag []
                  [Text "Log in as a demo user for example 'alice' (password alice)"]
```

```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world

delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who    = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice

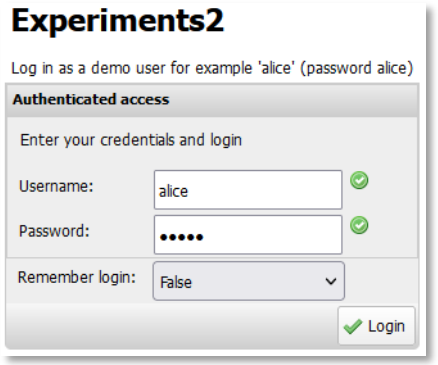
fred

```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world
```

```
delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who    = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice

fred

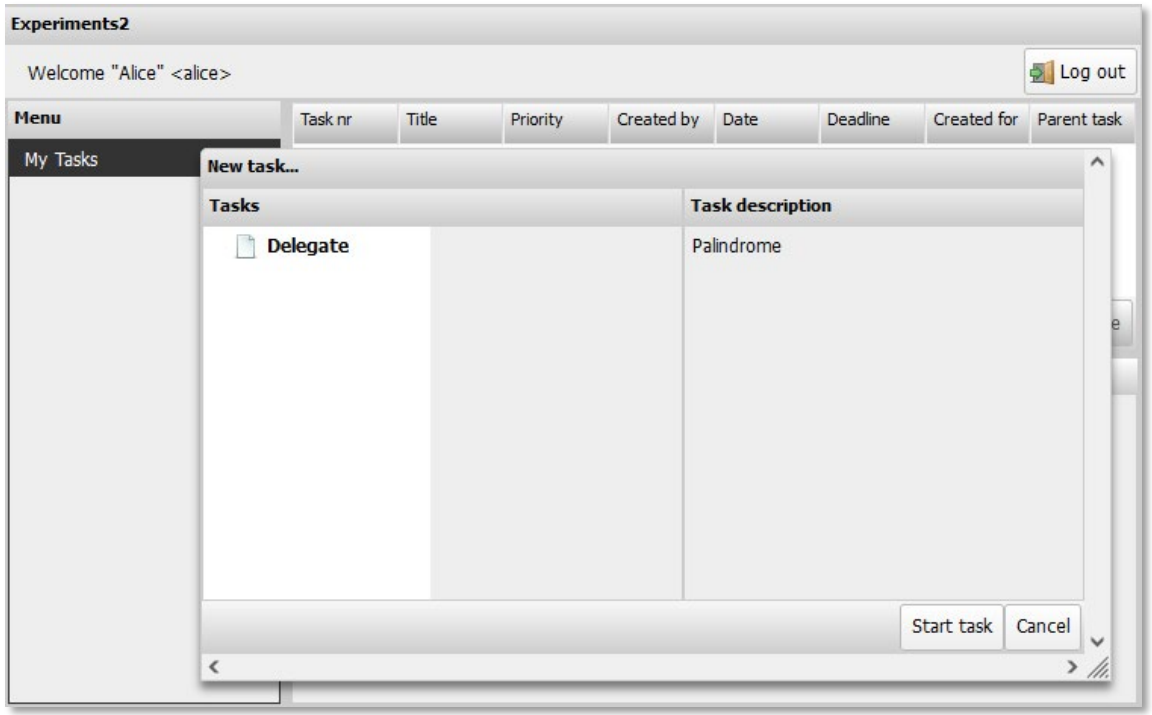


```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world
```

```
delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who     = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice

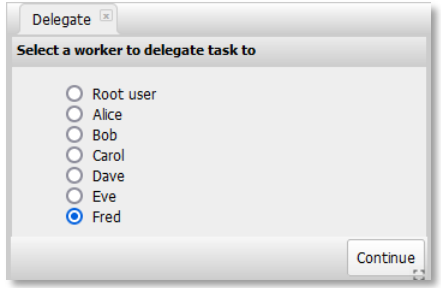
fred



```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world

delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who    = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice



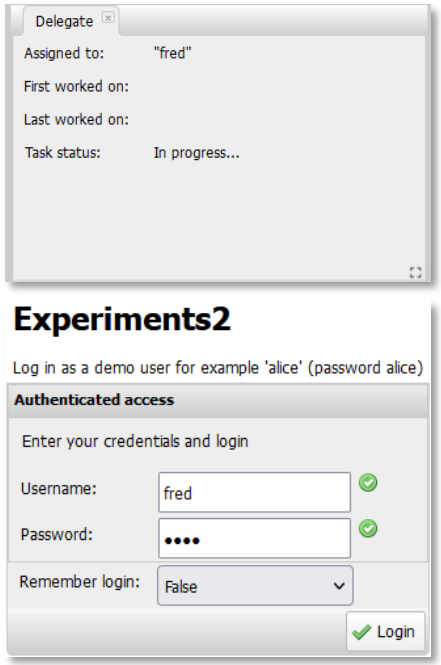
fred

```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world

delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who    = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice

fred

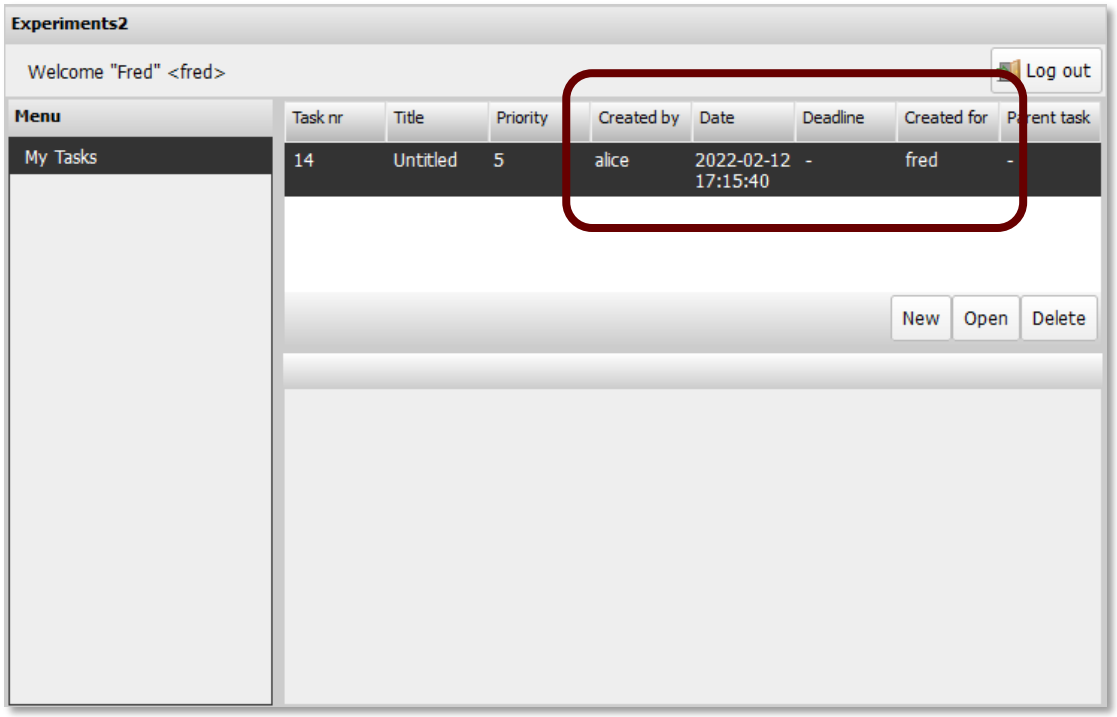


```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world
```

```
delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who    = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice

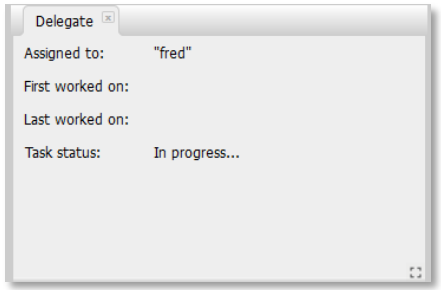
fred



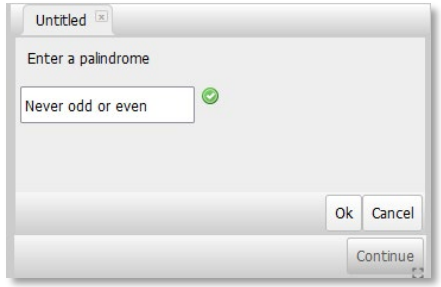
```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world

delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who    = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice



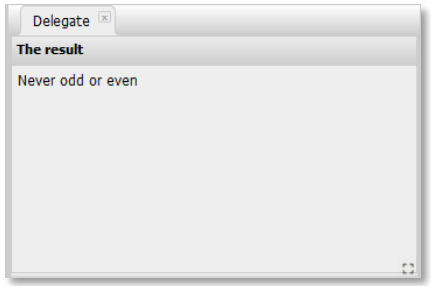
fred



```
Start :: *World -> *World
Start world
 = doTasks {… workflows = [workflow "Delegate" "Palindrome" (delegate palindrome)] …} world

delegate :: (Task a) -> Task a | iTask a
delegate task
 =                enterChoiceWithShared [ChooseFromCheckGroup id] users
                  <<@ Title "Select a worker to delegate task to"
   >>? \who     = who @: (task >>? return)
   >>- \result = viewInformation [] result <<@ Title "The result"
```

alice



fred

What have we done?

- Key points:
  - SDS: uniform, compositional, typed, interface to (persistent) information sources
  - Distributed Systems: assign tasks to workers / roles

```
@:(guest_lecture <<@ Hint "please applaud speaker")
```