HABITO

Habito: The Purely Functional Mortgage Broker

Will Jones, VP Engineering

HABITO

Facts and figures

- Founded in early 2015, live since early 2016
- Completely free service to take the pain out of mortgages
- Brokered over £1bn in applications to date
- ~140 people total, ~40 engineers
- Operate in small, cross-functional teams (~7 at present)

Old wounds

- No clear universal language
- Coupled inheritance hierarchies
- Complex runtime state
- Boilerplate

HABITO

New beginnings

- ~~No clear universal language~~
  Rich, data-driven domain model
- ~~Coupled inheritance hierarchies~~
  Compose simpler building blocks
- ~~Complex runtime state~~
  Immutability by default
- ~~Boilerplate~~
  Code generation from specifications

HABITO

Haskell

- Purely functional programming language
- Strong static typing
- Non-strict evaluation model
- Deploy binaries in Docker containers (for example)

Domain modelling

- “A transaction is either a purchase or a remortgage. A purchase involves a deposit and a property value. A remortgage involves a remaining balance, current monthly repayment and a property value.”

HABITO

Domain modelling

- “A transaction is either a purchase or a remortgage. A purchase involves a deposit and a property value. A remortgage involves a remaining balance, current monthly repayment and a property value.”

```
data Txn
  = Purchase PurchaseTxn
  | Remo RemoTxn
```

Domain modelling

- “A transaction is either a purchase or a remortgage. A purchase involves a deposit and a property value. A remortgage involves a remaining balance, current monthly repayment and a property value.”

```
data Txn
  = Purchase PurchaseTxn
  | Remo RemoTxn

data PurchaseTxn
  = PurchaseTxn
      { deposit :: GBP
      , propVal :: GBP
      }
```

Domain modelling

- “A transaction is either a purchase or a remortgage. A purchase involves a deposit and a property value. A remortgage involves a remaining balance, current monthly repayment and a property value.”

```
data Txn
  = Purchase PurchaseTxn
  | Remo RemoTxn
```

```
data PurchaseTxn
  = PurchaseTxn
      { deposit :: GBP
      , propVal :: GBP
      }
```

```
data RemoTxn
  = RemoTxn
      { balance     :: GBP
      , currMonthly :: GBP
      , propVal     :: GBP
      }
```

Domain modelling

- “A transaction is either a purchase or a remortgage. A purchase involves a deposit and a property value. A remortgage involves a remaining balance, current monthly repayment and a property value.”

```
data Txn
  = Purchase PurchaseTxn
  | Remo RemoTxn

data PurchaseTxn
  = PurchaseTxn
      { deposit :: GBP
      , propVal :: GBP
      }
```

```
txn1 :: Txn
txn1
  = Purchase (PurchaseTxn
      { deposit = 30000
      , propVal = 100000
      })
```

Domain modelling

- “Applicant credit policy rule: buy-to-let customers are not eligible for a mortgage if they are retired or will enter retirement before the end of the mortgage term.”

Domain modelling

- “Applicant credit policy rule: buy-to-let customers are not eligible for a mortgage if they are retired or will enter retirement before the end of the mortgage term.”

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
```

HABITO

Domain modelling

- “Applicant credit policy rule: buy-to-let customers are not eligible for a mortgage if they are retired or will enter retirement before the end of the mortgage term.”

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnScenario" .== buyToLet) $
      rejectIf $
```

Domain modelling

- “Applicant credit policy rule: buy-to-let customers are not eligible for a mortgage if they are retired or will enter retirement before the end of the mortgage term.”

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnScenario" .== buyToLet) $
      rejectIf $
        empType .== retired .||
          derive ageAtEndOfTerm .> retirementAge
```

Simpler building blocks

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnScenario" .== buyToLet) $
      rejectIf $
        empType .== retired .||
          derive ageAtEndOfTerm .> retirementAge
```

- Domain-specific language

HABITO

Simpler building blocks

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnScenario" .== buyToLet) $
      rejectIf $
        empType .== retired .||
          derive ageAtEndOfTerm .> retirementAge
```

- Domain-specific language
- Just a big function composition

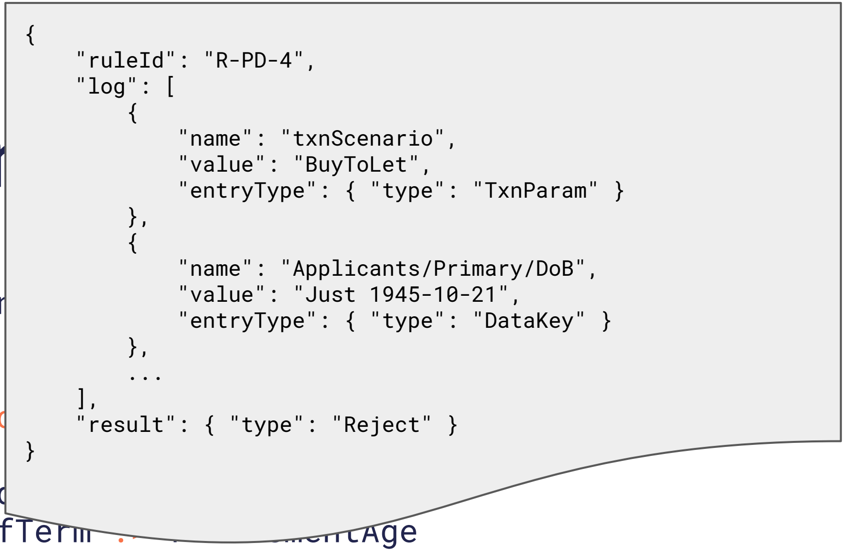
Simpler building blocks

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnScenario" .== buyToLet) $
      rejectIf $
        empType .== retired .||
          derive ageAtEndOfTerm .> retirementAge
```

- Domain-specific language
- Just a big function composition
- Some power tools: overloading, types

Simpler buildi

```
rPD4
  :: (HasToday, HasApplicar
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnSc
      rejectIf $
        empType .== retirec
          derive ageAtEndOfTerm              entAge
```



```
{
    "ruleId": "R-PD-4",
    "log": [
        {
            "name": "txnScenario",
            "value": "BuyToLet",
            "entryType": { "type": "TxnParam" }
        },
        {
            "name": "Applicants/Primary/DoB",
            "value": "Just 1945-10-21",
            "entryType": { "type": "DataKey" }
        },
        ...
    ],
    "result": { "type": "Reject" }
}
```

- Domain-specific language
- Just a big function composition
- Some power tools: overloading, types

Immutability everywhere

- Verify once, trust elsewhere
- Useful for parallelism/concurrency
- In general: easier to reason about
- Why stop with our language?

Data

profile

profile_id	account_id	first_name
3df81575-...	05d1100a-...	William

account

account_id	email	password	created	verified
05d1100a-...	will@example	<hash>	2018-11-22T..	t
dbc85161-...	dev@example	<hash>	2018-11-21T..	f

HABITO

Data

profile

profile_id	account_id	first_name
3df81575-...	05d1100a-...	William

account

account_id	email	password	created	verified
05d1100a-...	will@example	<hash>	2018-11-22T..	t
dbc85161-...	dev@example	<hash>	2018-11-21T..	t

Event sourcing

		Aggregate ID	Aggregate version	Aggregate type	Event data/payload

?	2018-11-21T...	05d1100a-...	1	Account	{
					“type”: “AccountCreated”,
					“value”: {
					“email”: “will@example”,
					“password”: “<hash>”
					}
					}

	2018-11-21T...	05d1100a-...	2	Account	{
					“type”: “PasswordChanged”,
					“value”: {
					“newPassword”: “<hash>”
					}
					}

	2018-11-22T...	05d1100a-...	3	Account	{
					“type”: “EmailVerified”,
					“value”: {}
					}

Event sourcing

- “An account is created. Thereafter the password may be changed, the email may be verified, ...”

```
data Account
  = Account
      { id  :: AccId
      , ...
      }

data AccEvent
  = Created { id :: AccId }
  | PassChanged { hashedPass :: HashedPass }
  | EmailVerified
  | ...
```

Event sourcing

```
data Maybe a = Nothing | Just a

updateAcc :: Maybe Account -> AccEvent -> Maybe Account
updateAcc (Just acc) (PassChanged pwd) =
  acc { password = pwd }
...

buildAcc
  :: [AccEvent]
  -> Maybe Account
buildAcc =
  foldl updateAcc Nothing
```

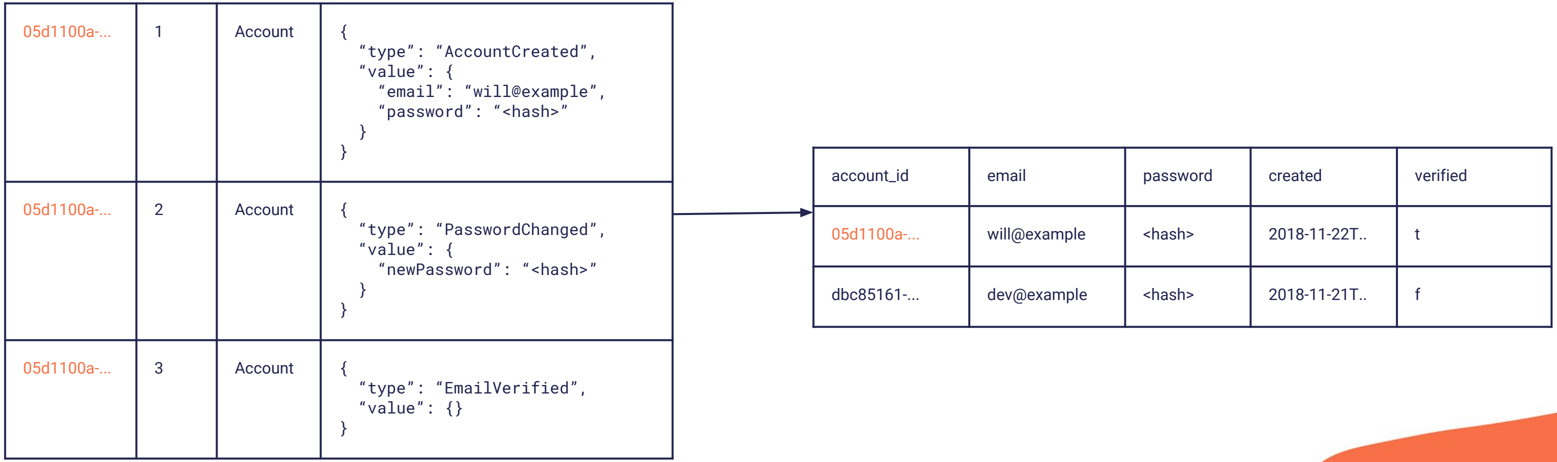
Event sourcing

```
data Maybe a = Nothing | Just a

updateAcc ::                                             nt
updateAcc
  acc {
...

buildAcc
  :: [AccEven
  -> Maybe
buildAcc
  foldl updateAcc Nothing
```

```
foldl f z [x1, x2, x3]
  == f (f (f z x1) x2) x3

foldl updateAcc Nothing [e1, e2, e3]
  == uA (uA (uA Nothing e1) e2) e3
```

HABITO

Asking questions



05d1100a-...	1	Account	{
			“type”: “AccountCreated”,
			“value”: {
			“email”: “will@example”,
			“password”: “<hash>”
			}
			}

05d1100a-...	2	Account	{
			“type”: “PasswordChanged”,
			“value”: {
			“newPassword”: “<hash>”
			}
			}

05d1100a-...	3	Account	{
			“type”: “EmailVerified”,
			“value”: {}
			}

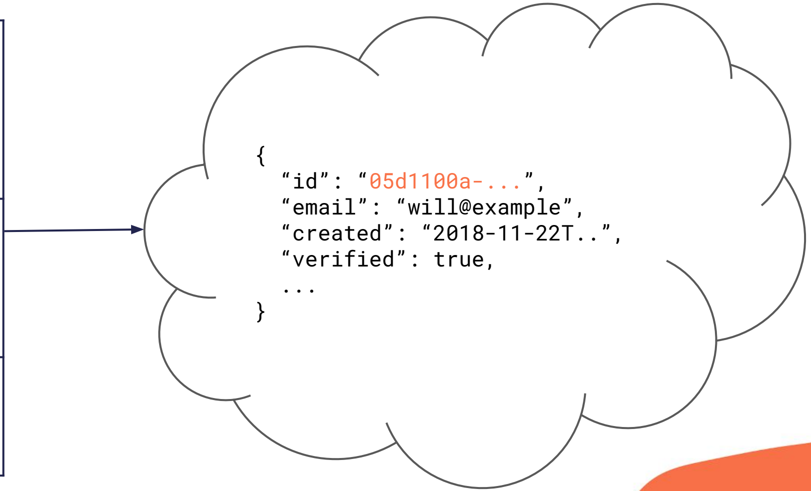
account_id	email	password	created	verified
05d1100a-...	will@example	<hash>	2018-11-22T..	t
dbc85161-...	dev@example	<hash>	2018-11-21T..	f

Asking questions

05d1100a-...	1	Account	{
			“type”: “AccountCreated”,
			“value”: {
			“email”: “will@example”,
			“password”: “<hash>”
			}
			}

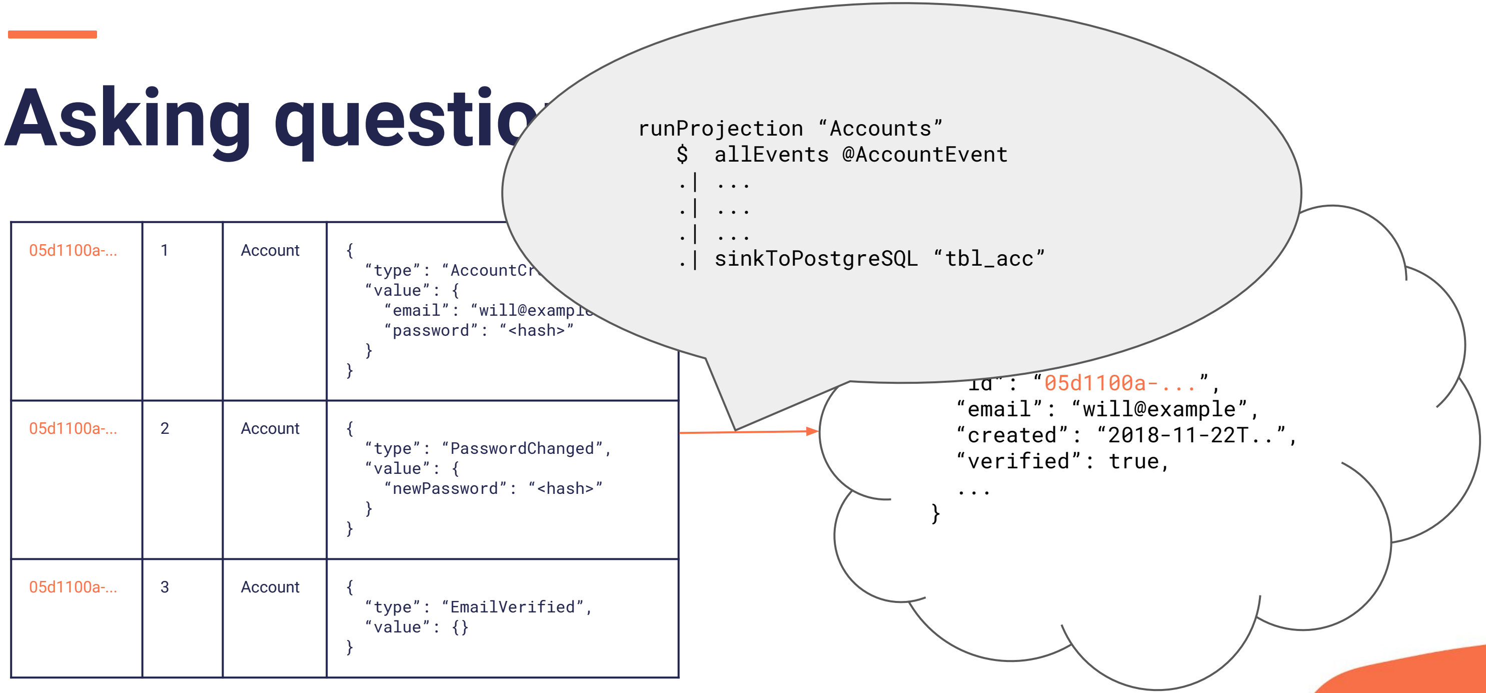
05d1100a-...	2	Account	{
			“type”: “PasswordChanged”,
			“value”: {
			“newPassword”: “<hash>”
			}
			}

05d1100a-...	3	Account	{
			“type”: “EmailVerified”,
			“value”: {}
			}



```
{
  “id”: “05d1100a-...”,
  “email”: “will@example”,
  “created”: “2018-11-22T..”,
  “verified”: true,
  ...
}
```

Asking questio



05d1100a-...	1	Account	{
			  “type”: “AccountCr
			  “value”: {
			    “email”: “will@example
			    “password”: “<hash>”
			  }
			}

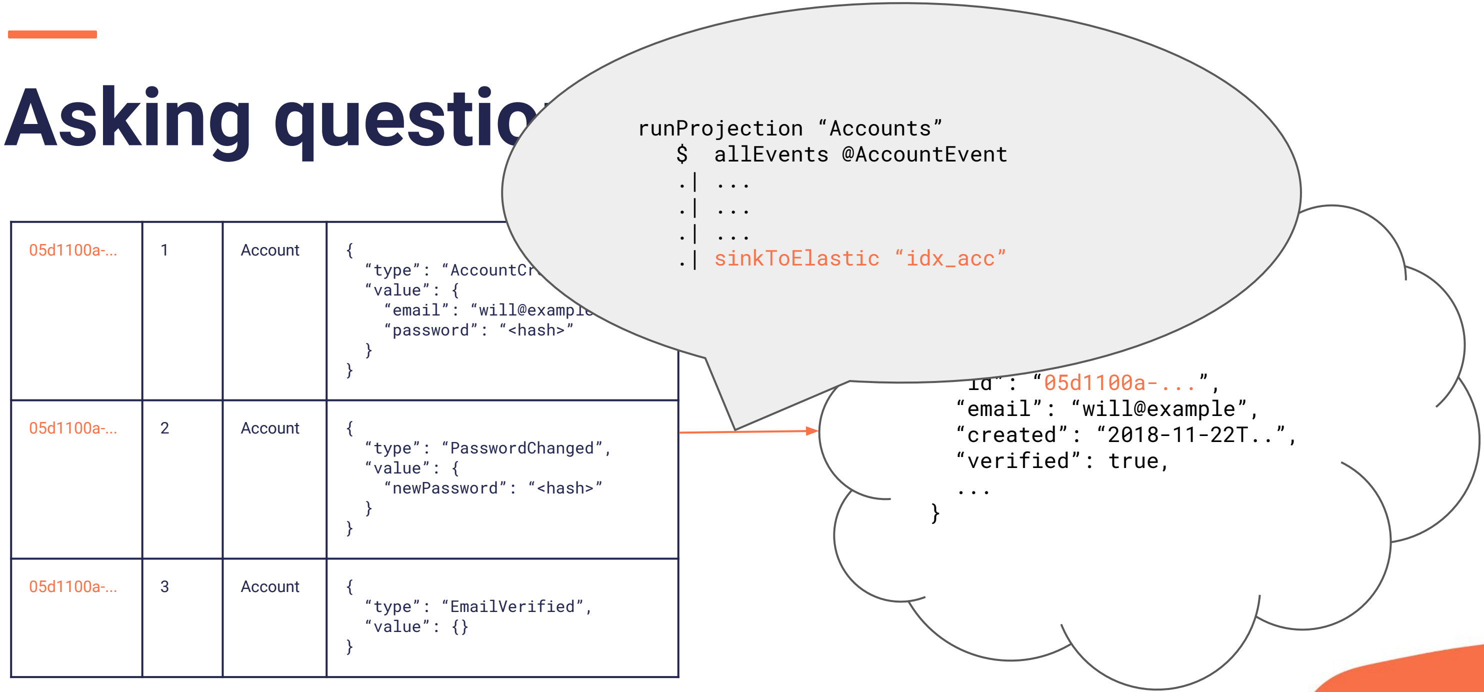
05d1100a-...	2	Account	{
			  “type”: “PasswordChanged”,
			  “value”: {
			    “newPassword”: “<hash>”
			  }
			}

05d1100a-...	3	Account	{
			  “type”: “EmailVerified”,
			  “value”: {}
			}

```
runProjection “Accounts”
   $  allEvents @AccountEvent
   .| ...
   .| ...
   .| ...
   .| sinkToPostgreSQL “tbl_acc”
```

```
  id”: “05d1100a-...”,
 “email”: “will@example”,
 “created”: “2018-11-22T..”,
 “verified”: true,
 ...
}
```

Asking questio



05d1100a-...	1	Account	{
			“type”: “AccountCr
			“value”: {
			“email”: “will@example
			“password”: “<hash>”
			}
			}

05d1100a-...	2	Account	{
			“type”: “PasswordChanged”,
			“value”: {
			“newPassword”: “<hash>”
			}
			}

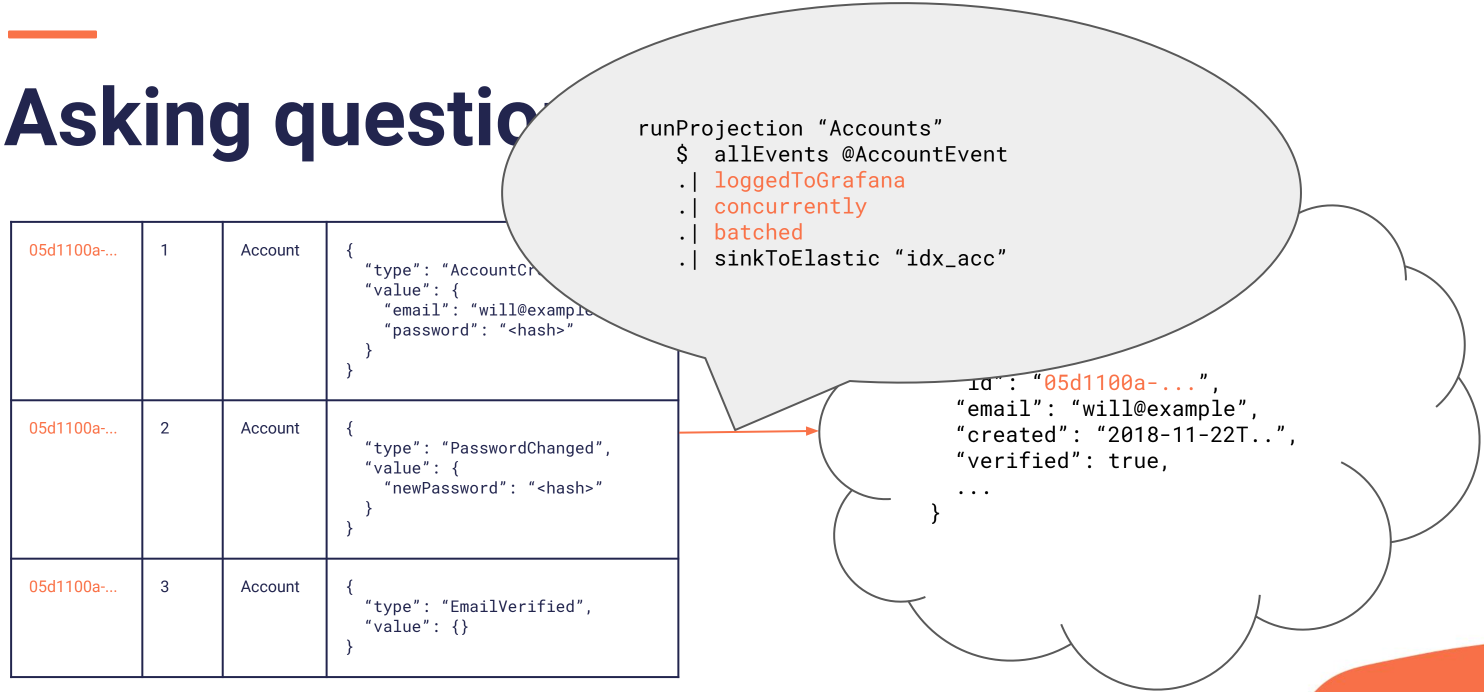
05d1100a-...	3	Account	{
			“type”: “EmailVerified”,
			“value”: {}
			}

```
runProjection “Accounts”
   $  allEvents @AccountEvent
   .| ...
   .| ...
   .| ...
   .| sinkToElastic “idx_acc”
```

```
   id”: “05d1100a-...”,
   “email”: “will@example”,
   “created”: “2018-11-22T..”,
   “verified”: true,
   ...
 }
```

HABITO

Asking questio



05d1100a-...	1	Account	{
			  “type”: “AccountCr
			  “value”: {
			    “email”: “will@example
			    “password”: “<hash>”
			  }
			}

05d1100a-...	2	Account	{
			  “type”: “PasswordChanged”,
			  “value”: {
			    “newPassword”: “<hash>”
			  }
			}

05d1100a-...	3	Account	{
			  “type”: “EmailVerified”,
			  “value”: {}
			}

```
runProjection “Accounts”
   $  allEvents @AccountEvent
  .| loggedToGrafana
  .| concurrently
  .| batched
  .| sinkToElastic “idx_acc”
```

```
   “id”: “05d1100a-...”,
   “email”: “will@example”,
   “created”: “2018-11-22T..”,
   “verified”: true,
   ...
 }
```

Boilerplate

```
data Txn
  = Purchase PurchaseTxn
  | Remo RemoTxn
```

```
data PurchaseTxn          data RemoTxn
  = PurchaseTxn             = RemoTxn
      { deposit :: GBP          { balance     :: GBP
      , propVal :: GBP          , currMonthly :: GBP
      }                         , propVal     :: GBP
                                }
```

Boilerplate

```
data Txn
  = Purchase PurchaseTxn
  | Remo RemoTxn
  deriving (Generic)
```

```
data PurchaseTxn
  = PurchaseTxn
      { deposit :: GBP
      , propVal :: GBP
      }
  deriving (Generic)
```

```
data RemoTxn
  = RemoTxn
      { balance     :: GBP
      , currMonthly :: GBP
      , propVal     :: GBP
      }
  deriving (Generic)
```

Boilerplate

```haskell
data Txn
  = Purchase PurchaseTxn
  | Remo RemoTxn
  deriving (Generic)
  deriving (FromJSON, ToJSON) via (Generically Txn)

data PurchaseTxn          data RemoTxn
  = PurchaseTxn             = RemoTxn
      { deposit :: GBP          { balance     :: GBP
      , propVal :: GBP          , currMonthly :: GBP
      }                         , propVal     :: GBP
  deriving (Generic)            }
                            deriving (Generic)
```

Boilerplate

```
updateTxnPropVal :: GBP -> Txn -> Txn
updateTxnPropVal x txn =
```

```
data PurchaseTxn          data RemoTxn
  = PurchaseTxn             = RemoTxn
      { deposit :: GBP              { balance     :: GBP
      , propVal :: GBP              , currMonthly :: GBP
      }                             , propVal     :: GBP
  deriving (Generic)                }
                            deriving (Generic)
```

HABITO

Boilerplate

```
updateTxnPropVal :: GBP -> Txn -> Txn
updateTxnPropVal x txn =
  case txn of
    PurchaseTxn pTxn -> ...
    RemoTxn     rTxn -> ...

data PurchaseTxn          data RemoTxn
  = PurchaseTxn             = RemoTxn
      { deposit :: GBP          { balance     :: GBP
      , propVal :: GBP          , currMonthly :: GBP
      }                         , propVal     :: GBP
  deriving (Generic)            }
                            deriving (Generic)
```

Boilerplate

```
updateTxnPropVal :: GBP -> Txn -> Txn
updateTxnPropVal x txn =
  set (nestedField @“propVal”) x txn
```

```
data PurchaseTxn
  = PurchaseTxn
      { deposit :: GBP
      , propVal :: GBP
      }
  deriving (Generic)
```

```
data RemoTxn
  = RemoTxn
      { balance     :: GBP
      , currMonthly :: GBP
      , propVal     :: GBP
      }
  deriving (Generic)
```

Compiling domains

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnScenario" .== buyToLet) $ ...
```

```
{
    "ruleId": "R-PD-4",
    "log": [
        {
            "name": "txnScenario",
            "value": "BuyToLet",
            "entryType": { "type": "TxnParam" }
        },
        ...
    ],
    "result": { "type": "Reject" }
}
```

Compiling domains

```
rPD4
  :: (HasToday, HasApplicant)
  => RuleBuilder "R-PD-4"
rPD4 _
  = given (txnParam @"txnScenario" .== buyToLet) $ ...
```

```
{
    "ruleId": "R-PD-4",
    "log": [
        {
            "name": "txnScenario",
            "value": "BuyToLet",
            "entryType": { "type": "TxnParam" }
        },
        ...
    ],
    "result": { "type": "Reject" }
}
```

HABITO

It can’t all be good news

- Type-checking and compilation times
- Laziness and reasoning about performance
- Language ecosystem -- tooling and libraries
- Recruitment and hiring
- Event sourcing has its own challenges -- reprojection

HABITO

Useful underpinnings

- ~~No clear universal language~~
  Rich, data-driven domain model
- ~~Coupled inheritance hierarchies~~
  Compose simpler building blocks
- ~~Complex runtime state~~
  Immutability by default
- ~~Boilerplate~~
  Code generation from specifications

HABITO

Thank you