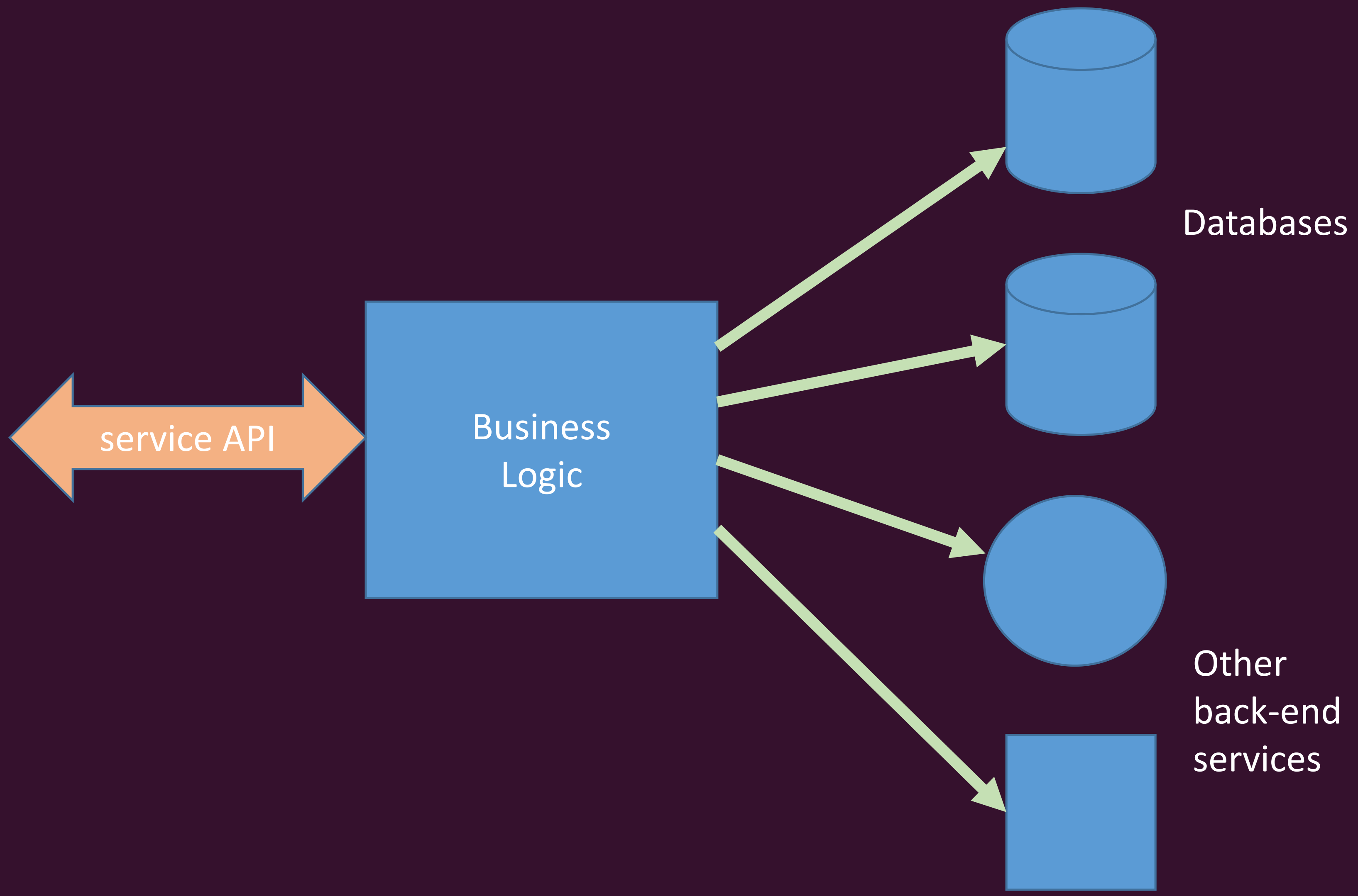
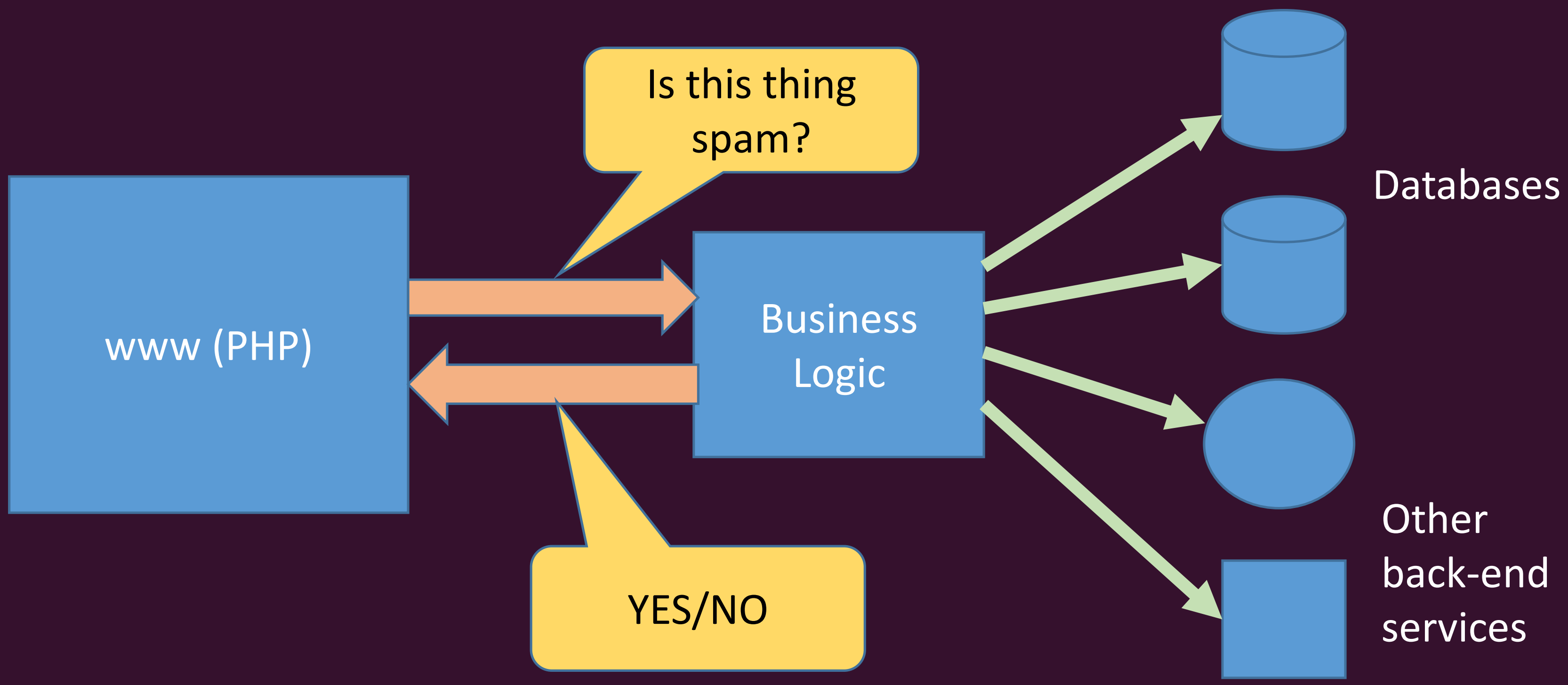
The Haxl Project at Facebook

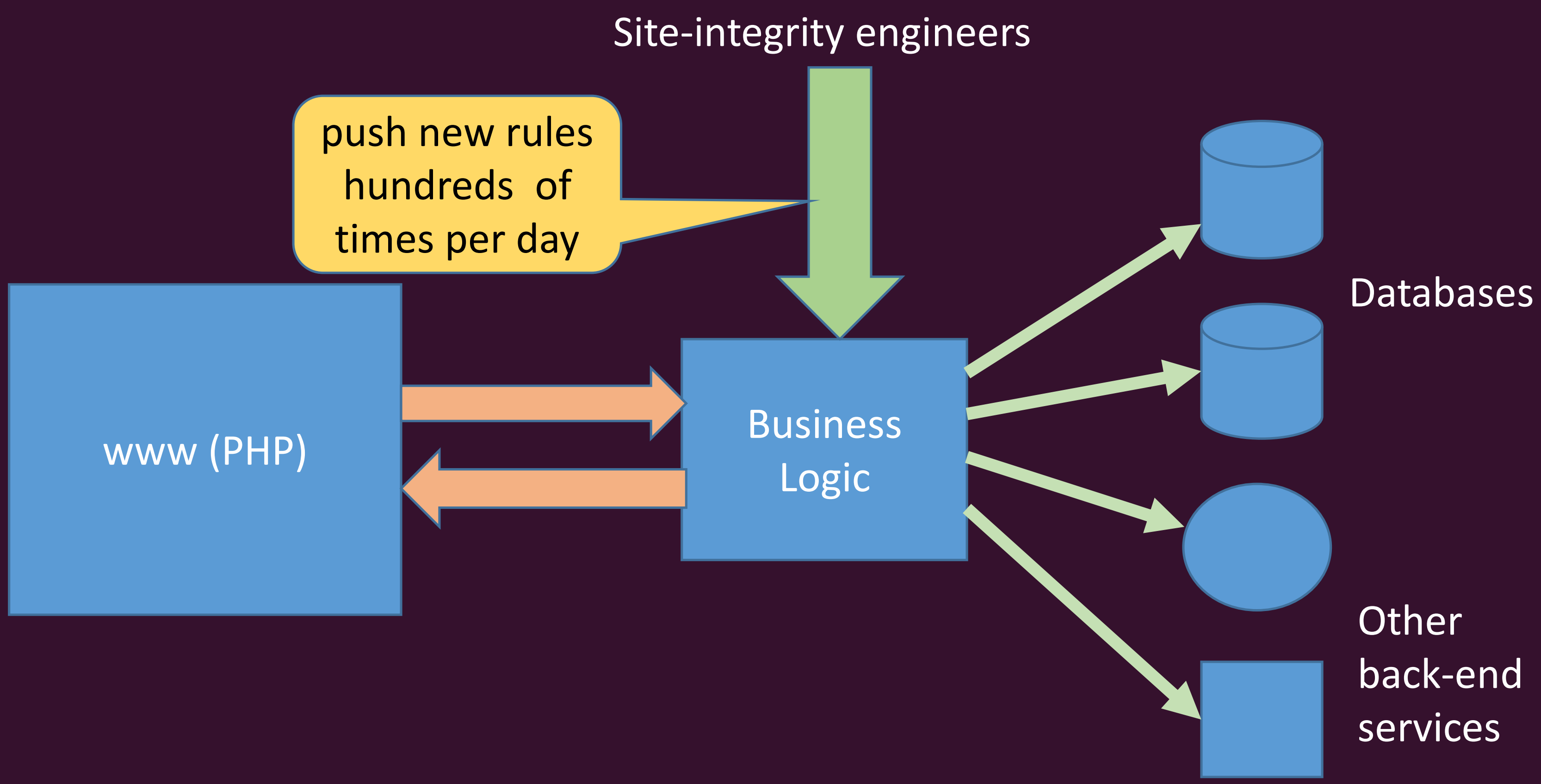
Simon Marlow
Jon Coens
Louis Brandy
Jon Purdy
& others


service API
Business Logic
Databases
Other back-end services

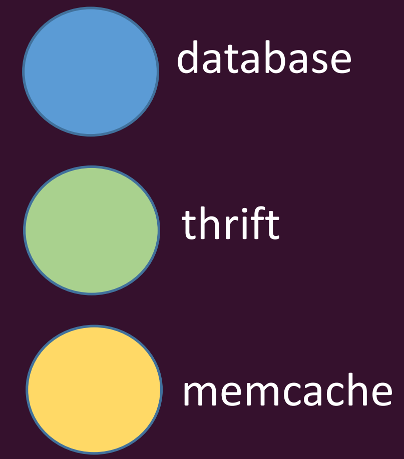
Use case: fighting spam

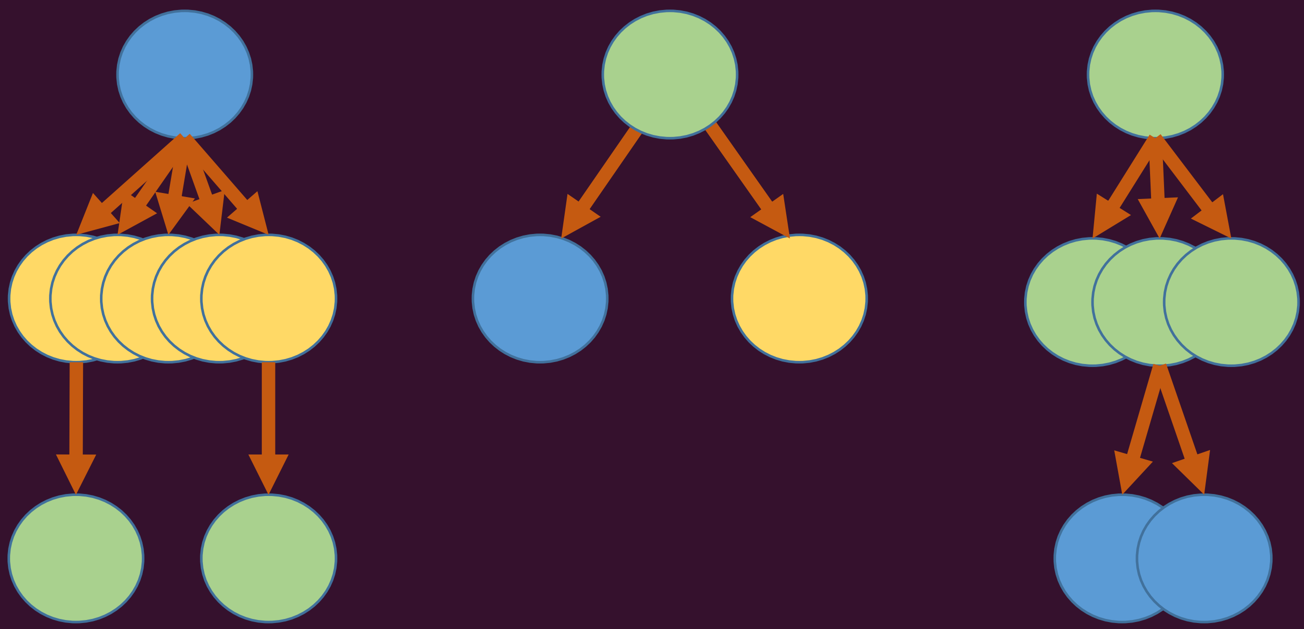


Use case: fighting spam



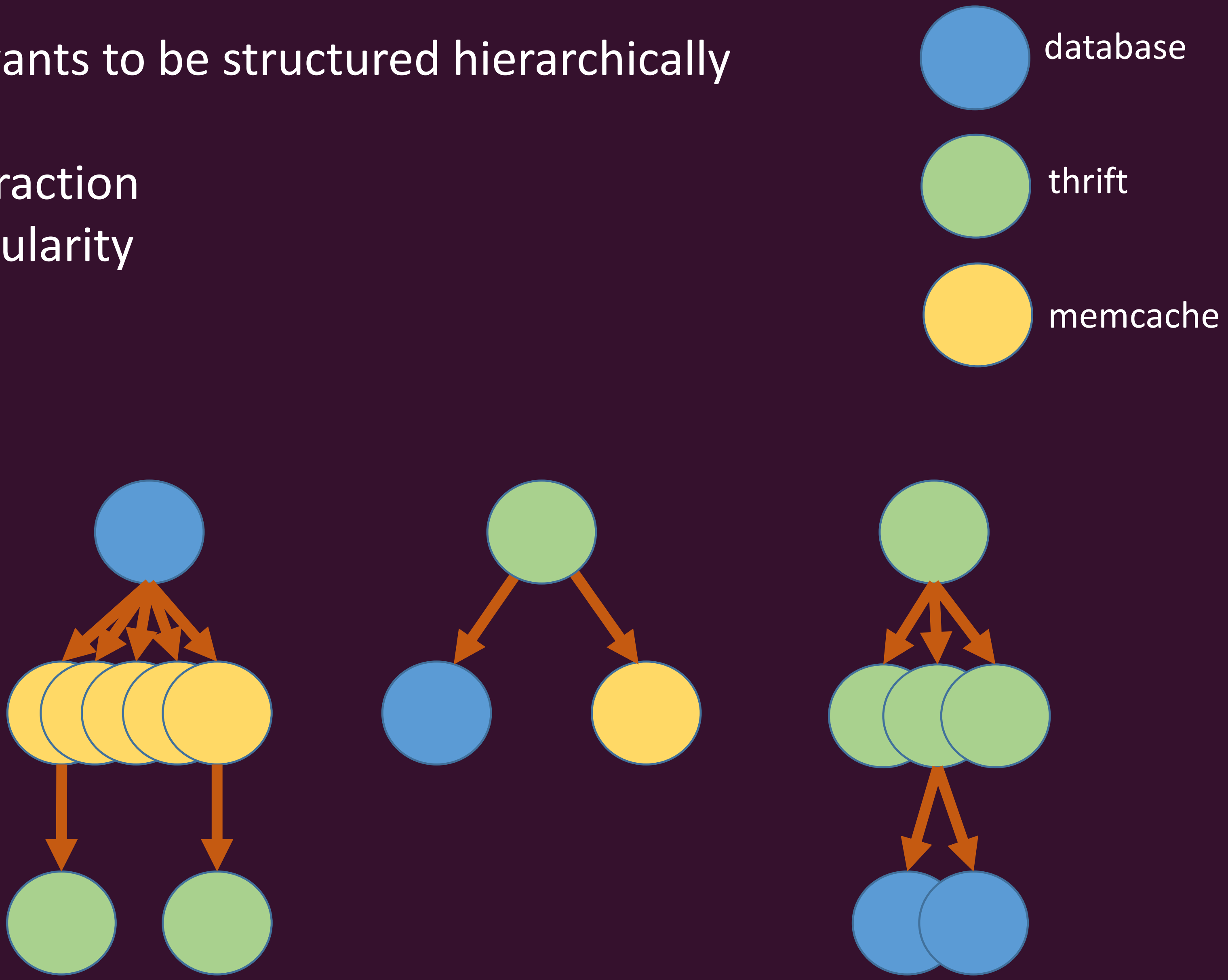
Data dependencies in a computation





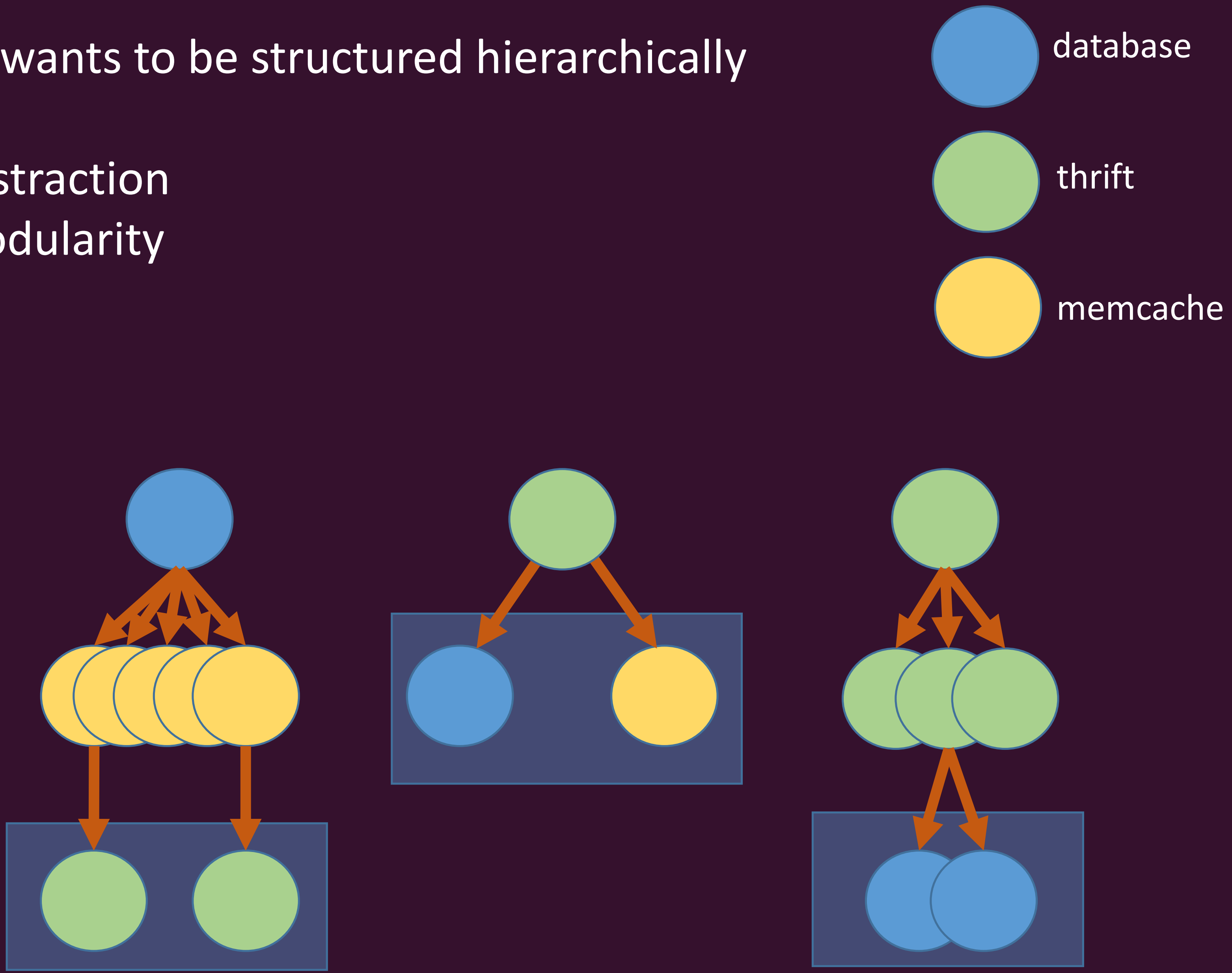
Code wants to be structured hierarchically

- abstraction
- modularity



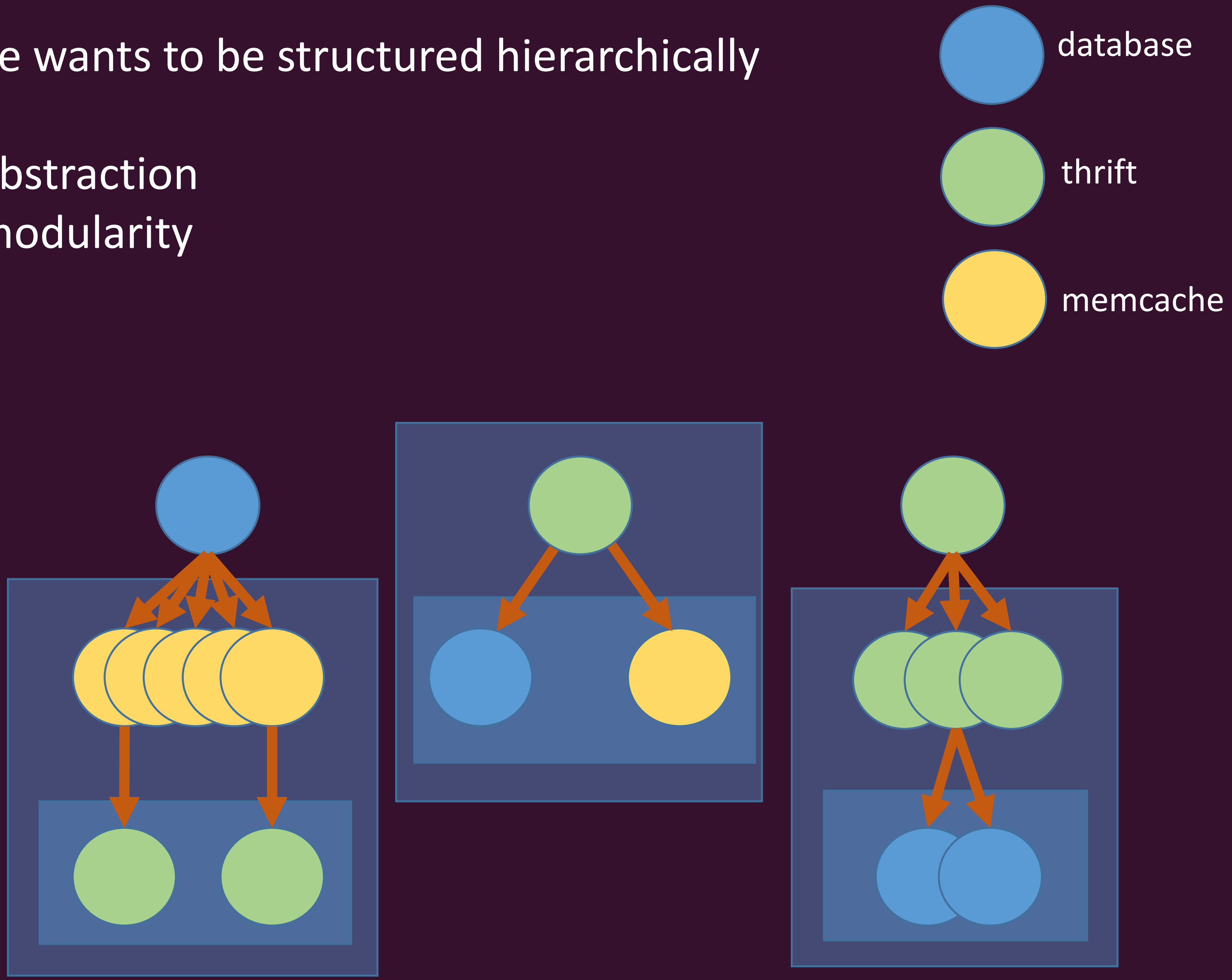
Code wants to be structured hierarchically

- abstraction
- modularity



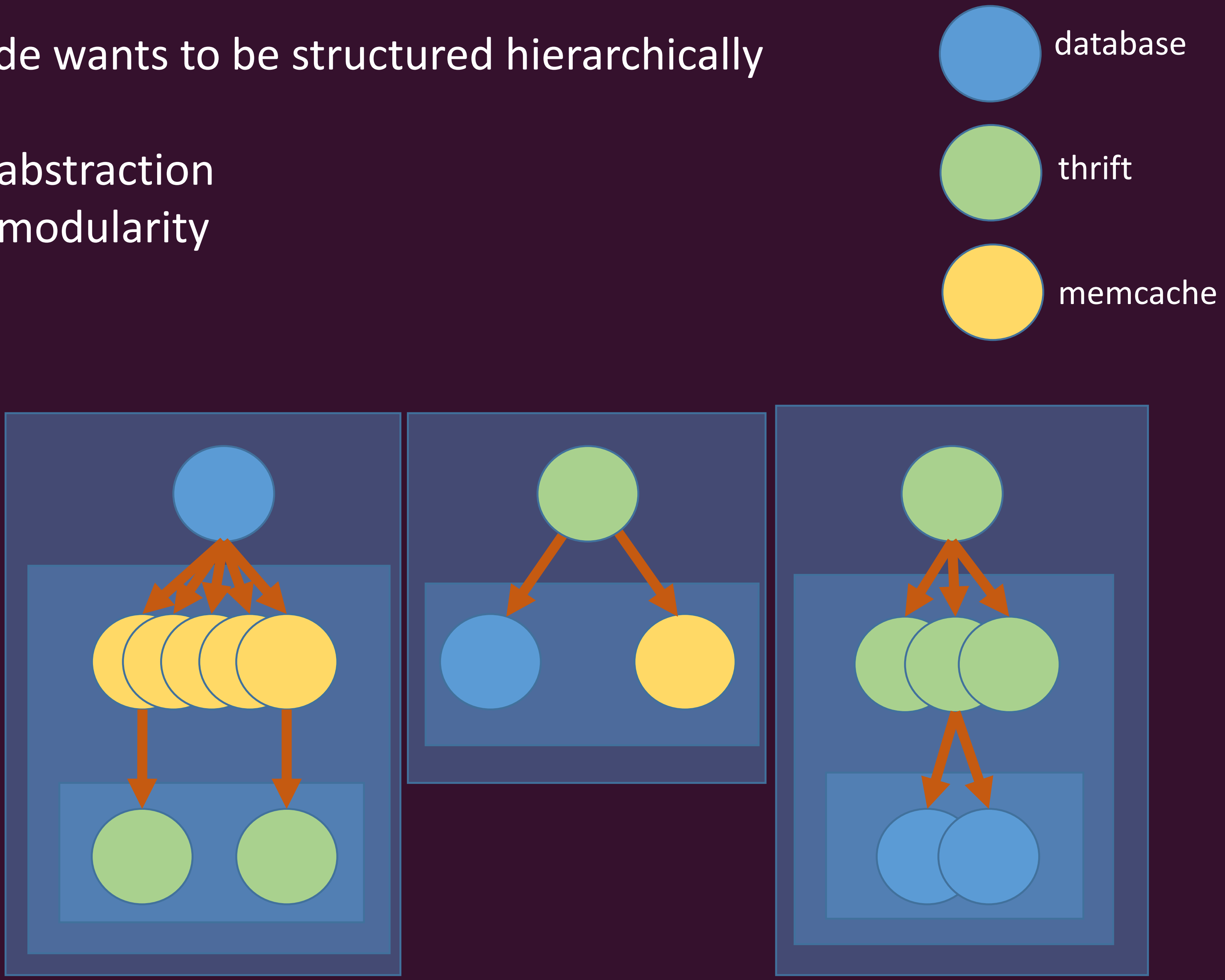
Code wants to be structured hierarchically

- abstraction
- modularity



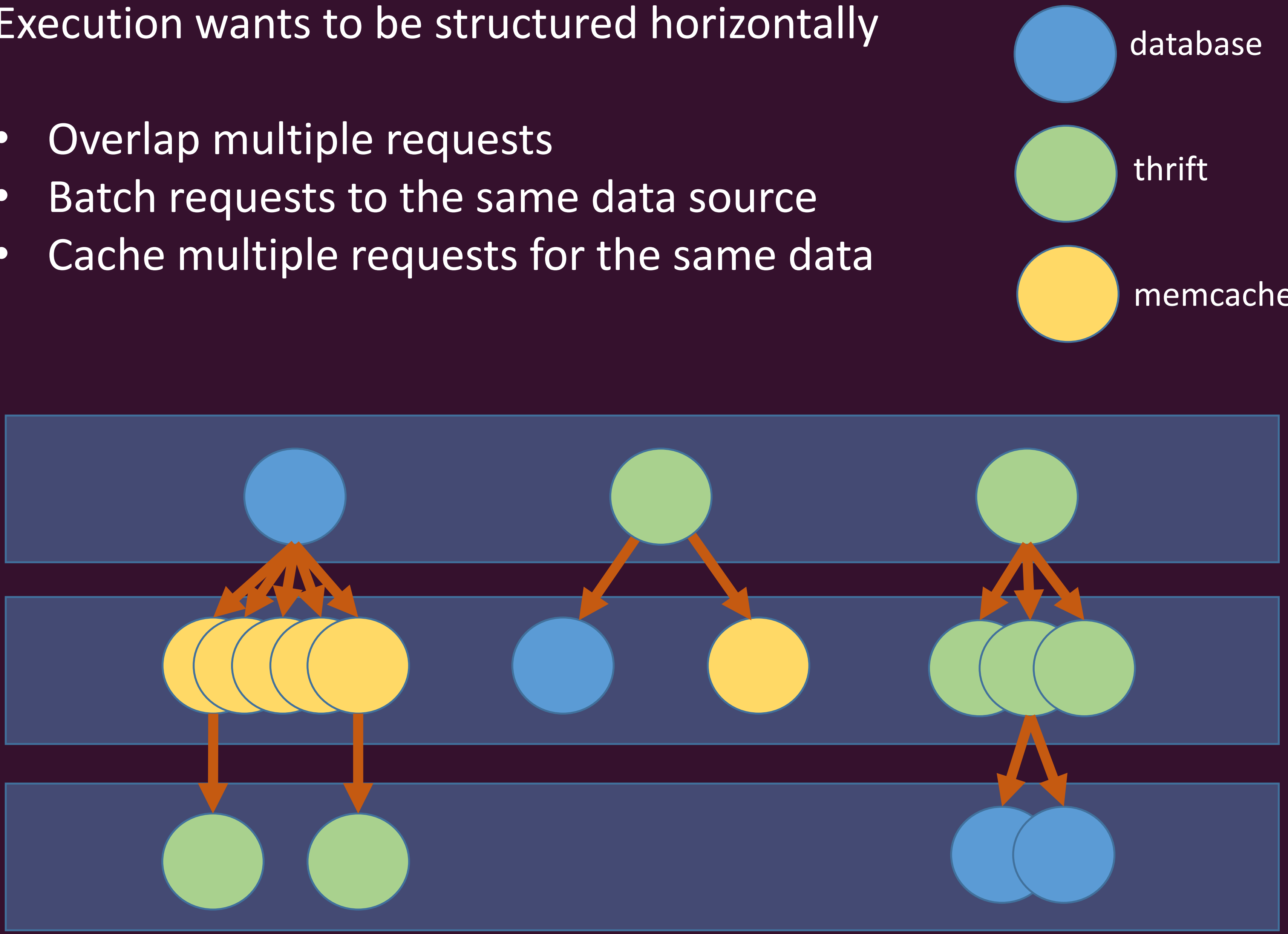
Code wants to be structured hierarchically

- abstraction
- modularity



Execution wants to be structured horizontally

- Overlap multiple requests
- Batch requests to the same data source
- Cache multiple requests for the same data



- Furthermore, each data source has different characteristics
  - Batch request API?
  - Sync or async API?
  - Set up a new connection for each request, or keep a pool of connections around?

- Want to abstract away from *all* of this in the business logic layer

But we know how to do this!

But we know how to do this!

- Concurrency.

  Threads let us keep our abstractions & modularity while executing things at the same time.
  - Caching/batching can be implemented as a service in the process
    - as we do with the IO manager in GHC

But we know how to do this!

- Concurrency.
  Threads let us keep our abstractions & modularity while executing things at the same time.
  - Caching/batching can be implemented as a service in the process
    - as we do with the IO manager in GHC

- But concurrency (the programing model) isn’t what we want here.

But we know how to do this!

- Concurrency.

    Threads let us keep our abstractions & modularity while executing things at the same time.

    - Caching/batching can be implemented as a service in the process
        - as we do with the IO manager in GHC

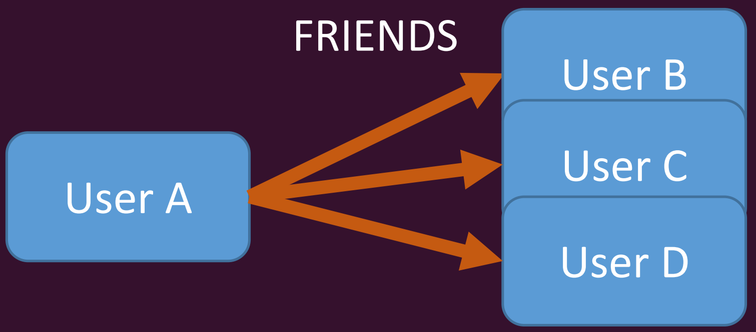
- But concurrency (the programing model) isn't what we want here.
- Example...

- x and y are Facebook users
- suppose we want to compute the number of friends that x and y have in common
- simplest way to write this:

```
length (intersect (friendsOf x) (friendsOf y))
```

Brief detour: TAO

- TAO implements Facebook's data model
  - most important data source we need to deal with
- Data is a graph
  - Nodes are "objects", identified by 64-bit ID
  - Edges are "assocs" (directed; a pair of 64-bit IDs)
- Objects and assocs have a type
  - object fields determined by the type
- Basic operations:
  - Get the object with a given ID
  - Get the assocs of a given type from a given ID



- Back to our example

```
length (intersect (friendsOf x) (friendsOf y))
```

- (friendsOf x) makes a request to TAO to get all the IDs for which there is an assoc of type FRIEND (x,_).
- TAO has a multi-get API; very important that we submit (friendsOf x) and (friendsOf y) as a single operation.

Using concurrency

- This:

```
length (intersect (friendsOf x) (friendsOf y))
```

Using concurrency

- This:

```
length (intersect (friendsOf x) (friendsOf y))
```

- Becomes this:

```
do
   m1 <- newEmptyMVar
   m2 <- newEmptyMVar
   forkIO (friendsOf x >>= putMVar m1)
   forkIO (friendsOf y >>= putMVar m2)
   fx <- takeMVar m1
   fy <- takeMVar m2
   return (length (intersect fx fy))
```


OH GOD
MY EYES

- Using the async package:

```
do
     ax <- async (friendsOf x)
     ay <- async (friendsOf y)
     fx <- wait ax
     fy <- wait ay
     return (length (intersect fx fy))
```

- Using Control.Concurrent.Async.concurrently:

```
do
      (fx,fy) <- concurrently (friendsOf x) (friendsOf y)
      return (length (intersect fx fy))
```

Why not concurrency?

- friendsOf x and friendsOf y are
  - obviously independent
  - obviously both needed
  - “pure”

Why not concurrency?

- friendsOf x and friendsOf y are
  - obviously independent
  - obviously both needed
  - “pure”

- Caching is not just an optimisation:
  - if friendsOf x is requested twice, we *must* get the same answer both times
  - *caching is a requirement*

Why not concurrency?

- friendsOf x and friendsOf y are
  - obviously independent
  - obviously both needed
  - “pure”
- Caching is not just an optimisation:
  - if friendsOf x is requested twice, we *must* get the same answer both times
  - *caching is a requirement*
- we don’t want the programmer to have to ask for concurrency here

- Could we use unsafePerformIO?

```
  length (intersect (friendsOf x) (friendsOf y))

friendsOf = unsafePerformIO ( .. )
```

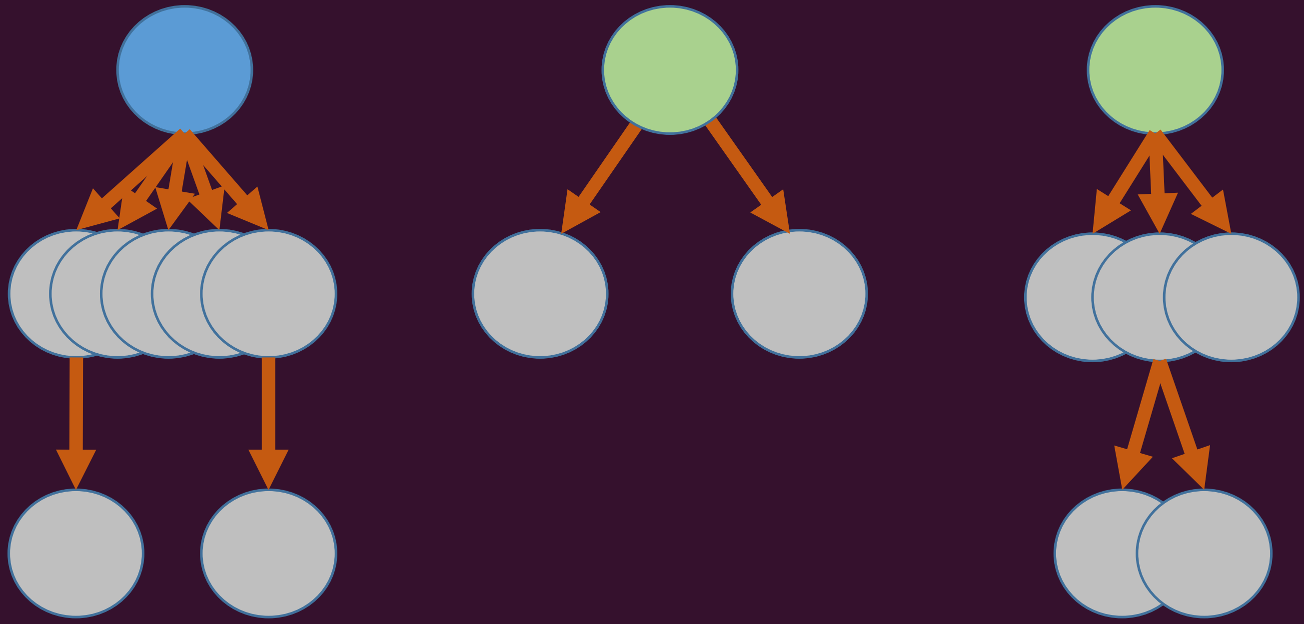
- we could do caching this way, but not concurrency. Execution will stop at the first data fetch.

Central problem

- Reorder execution of an expression to perform data fetching optimally.
- The programming model has no side effects (other than reading)

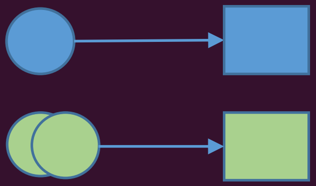
What we would like to do:

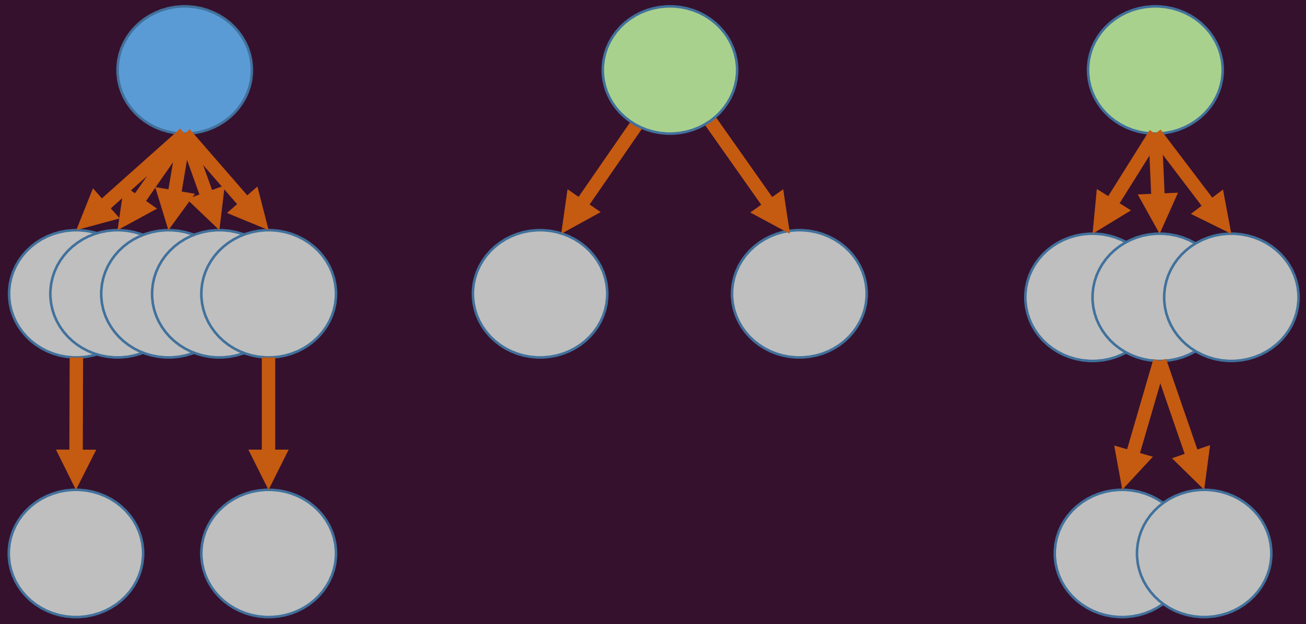
- explore the expression along all branches to get a set of data fetches



What we would like to do:

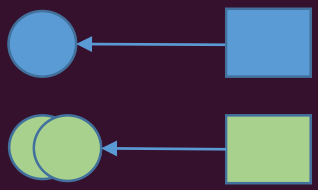
- submit the data fetches

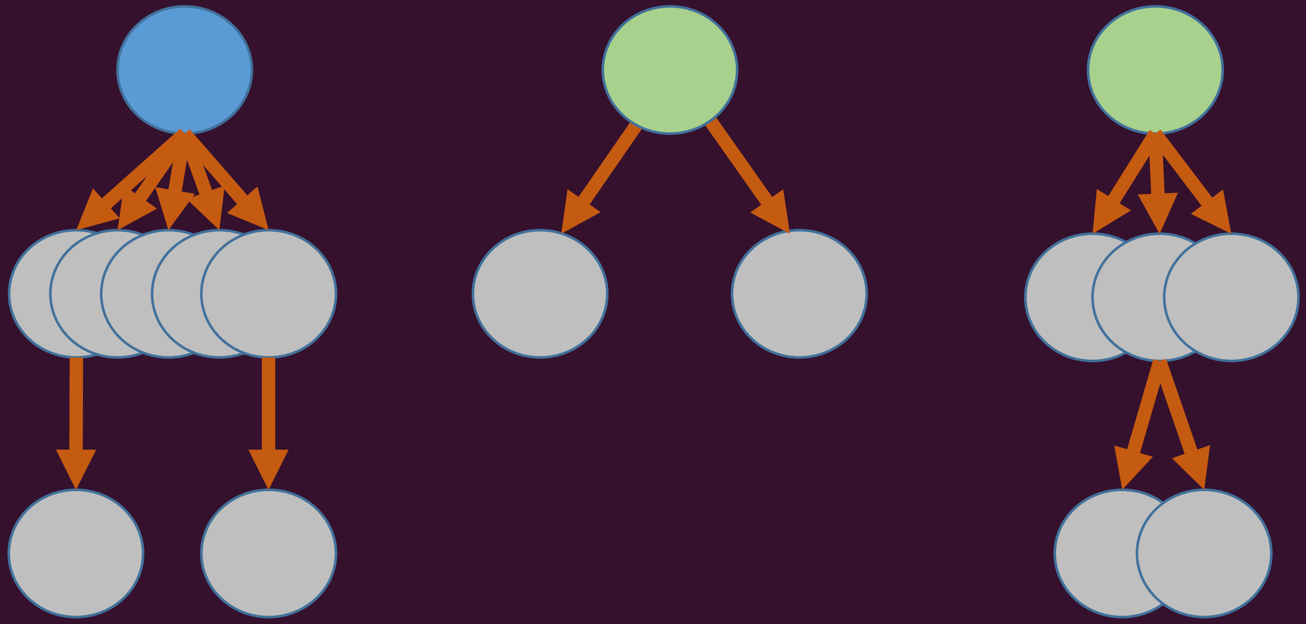




What we would like to do:

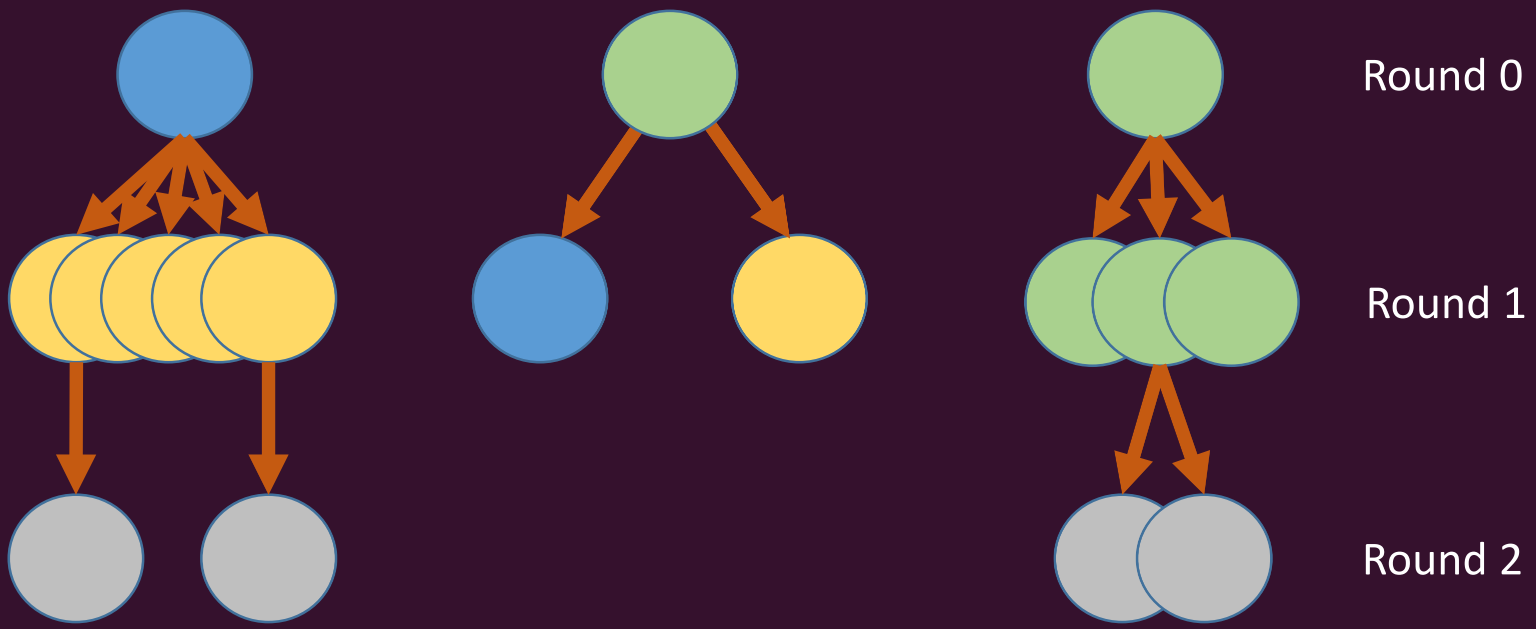
- wait for the responses





What we would like to do:

- now the computation is unblocked along multiple paths
- ... explore again
- collect the next batch of data fetches
- and so on



Fighting spam with pure functions

By Louis Brandy on Thursday, 24 January 2013 at 17:31

Like any popular Internet site, Facebook is a target for abuse. Our Site Integrity engineers rely on FXL, a domain-specific language forged in the fires of spam fighting at Facebook, to

- Facebook’s existing solution to this problem: FXL
- Lets you write

```
Length(Intersect(FriendsOf(X),FriendsOf(Y)))
```

- And optimises the data fetching correctly.
- But it’s an interpreter, and works with an explicit representation of the computation graph.

- We want to run compiled code for efficiency
- And take advantage of Haskell
  - high quality implementation
  - great libraries for writing business logic etc.

- So, how can we implement the right data fetching behaviour in a Haskell DSL?

Start with a concurrency monad

```haskell
newtype Haxl a = Haxl { unHaxl :: Result a }

data Result a = Done a
              | Blocked (Haxl a)

instance Monad Haxl where
  return a = Haxl (Done a)
  m >>= k = Haxl $
    case unHaxl m of
      Done a    -> unHaxl (k a)
      Blocked r -> Blocked (r >>= k)
```

Start with a concurrency monad

```haskell
newtype Haxl a = Haxl { unHaxl :: Result a }

data Result a = Done a
              | Blocked (Haxl a)

instance Monad Haxl where
  return a = Haxl (Done a)
  m >>= k = Haxl $
    case unHaxl m of
      Done a    -> unHaxl (k a)
      Blocked r -> Blocked (r >>= k)
```

It’s a
Free
Monad

- The concurrency monad lets us run a computation until it blocks, do something, then resume it
- But we need to know what it blocked on...
- Could add some info to the Blocked constructor

```haskell
newtype Haxl a = Haxl { unHaxl :: Responses -> Result a }

data Result a = Done a
              | Blocked Requests (Haxl a)

instance Monad Haxl where
  return a = Haxl $ \_ -> Done a
  Haxl m >>= k = Haxl $ \resps ->
    case m resps of
      Done a         -> unHaxl (k a) resps
      Blocked reqs r -> Blocked reqs (r >>= k)

addRequest    :: Request a -> Requests -> Requests
emptyRequests :: Requests

fetchResponse :: Request a -> Responses -> a

dataFetch :: Request a -> Haxl a
dataFetch req = Haxl $ \_ ->
  Blocked (addRequest req emptyRequests) $ Haxl $ \resps ->
    Done (fetchResponse req resps)
```

- Ok so far, but we still get blocked at the first data fetch.

```
numCommonFriends x y = do
   fx <- friendsOf x
   fy <- friendsOf y
   return (length (intersect fx fy))
```

Blocked here

- To explore multiple branches, we need to use Applicative

```
instance Applicative Haxl where

  pure = return

  Haxl f <*> Haxl a = Haxl $ \resps ->
    case f resps of
      Done f' ->
        case a resps of
          Done a'           -> Done (f' a')
          Blocked reqs a' -> Blocked reqs (f' <$> a')
      Blocked reqs f' ->
        case a resps of
          Done a'            -> Blocked reqs (f' <*> return a')
          Blocked reqs' a' -> Blocked (reqs <> reqs') (f' <*> a')
```

<*> :: Applicative f => f (a -> b) -> f a -> f b

- This is precisely the advantage of Applicative over Monad:
  - Applicative allows exploration of the structure of the computation

- Our example is now written:

```
numCommonFriends x y =
  length <$> (intersect <$> friendsOf x <*> friendsOf y)
```

- Or:

```
numCommonFriends x y =
  length <$> common (friendsOf x) (friendsOf y)
  where common = liftA2 intersect
```

- Note that we still have the Monad!
- The Monad allows us to make decisions based on values when we need to.

```
do
  fs <- friendsOf x
  if simon `elem` fs
     then ...
     else ...
```

Blocked here

- Batching will not explore the then/else branches
  - exactly what we want.

- But it does mean the programmer should use Applicative composition to get batching.
- This is suboptimal:

```
do
  fx <- friendsOf x
  fy <- friendsOf y
  return (length (intersect fx fy))
```

- So our plan is to
  - provide APIs that batch correctly
  - translate do-notation into Applicative where possible
    - (forthcoming GHC extension)

- We really want bulk operations to benefit from batching.

```
friendsOfFriends id =
  concat <$> (mapM friendsOf =<< friendsOf id)
```

- But this doesn't work: mapM uses Monad rather than Applicative composition.
- This is why traverse exists:

```
traverse :: (Traversable t, Applicative f)
         => (a -> f b) -> t a -> f (t b)
```

- So in our library, we make mapM = traverse
- Also: sequence = sequenceA
- Will be fixed once Applicative is a superclass of Monad

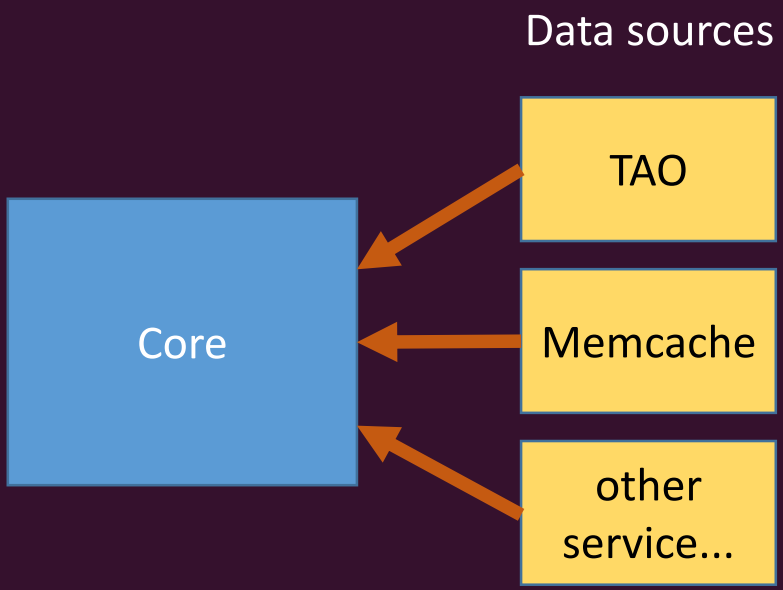
Implementation

- DataSource abstraction
- Replaying requests
- Scaling
- Hot-code swapping

- Experience
- Status etc.

Data Source Abstraction

- We want to structure the system like this:



- Core code includes the monad, caching support etc.
- Core is *generic:* no data sources built-in

How do we arrange this?

- Three ways that a data source interacts with core:
  - issuing a data fetch request
  - persistent state
  - fetching the data

- Package this up in a type class

```
class DataSource req where
  ...
```

parameterised over the type of *requests*

- Let’s look at requests first...

Example Request type

```
data ExampleReq a where
  CountAardvarks :: String -> ExampleReq Int
  ListWombats    :: Id     -> ExampleReq [Id]
  deriving Typeable
```

it's a GADT, where the type parameter is the type of the result of this request

- Core has a single way to issue a request

```
dataFetch :: DataSource req => req a -> Haxl a
```

- Note how the result type matches up.

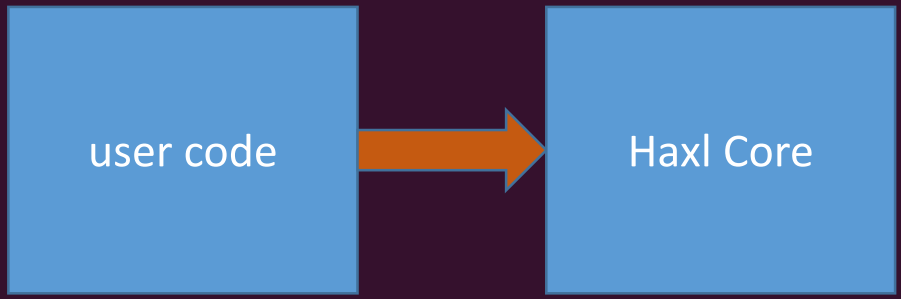
- Clean data source abstraction

- Means that we can plug in any set of data sources at *runtime*
  - e.g. mock data sources for testing and experimentation
  - core code can be built & tested independently

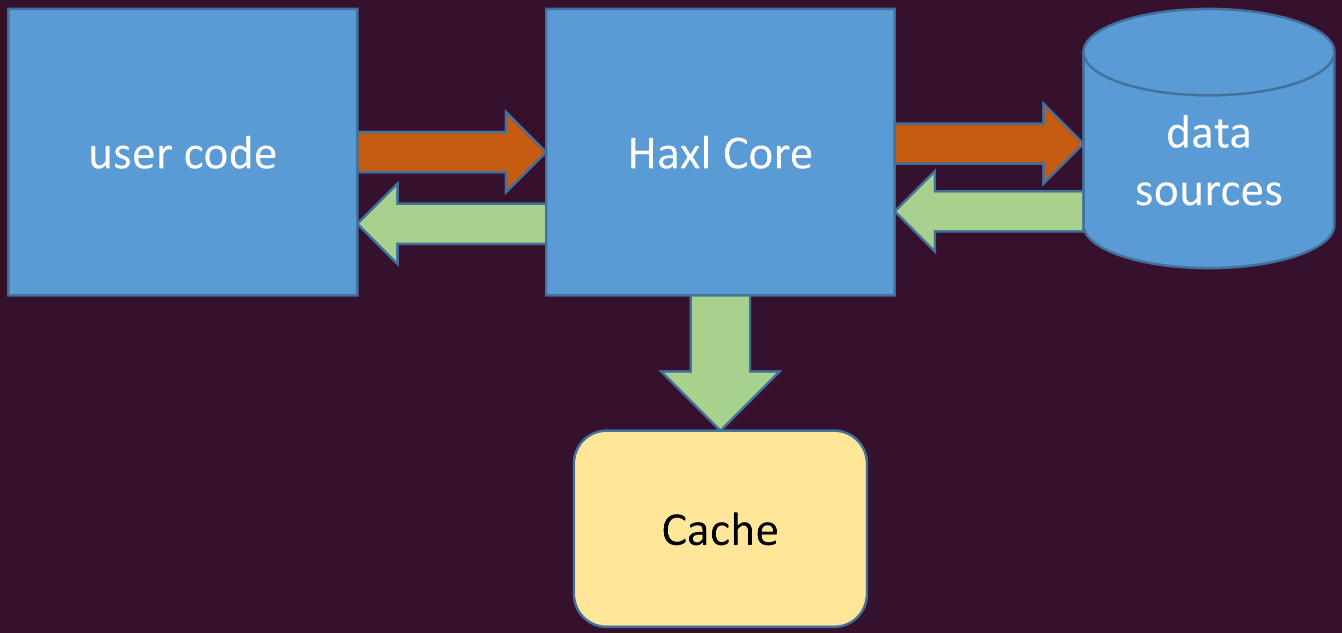
Replayability

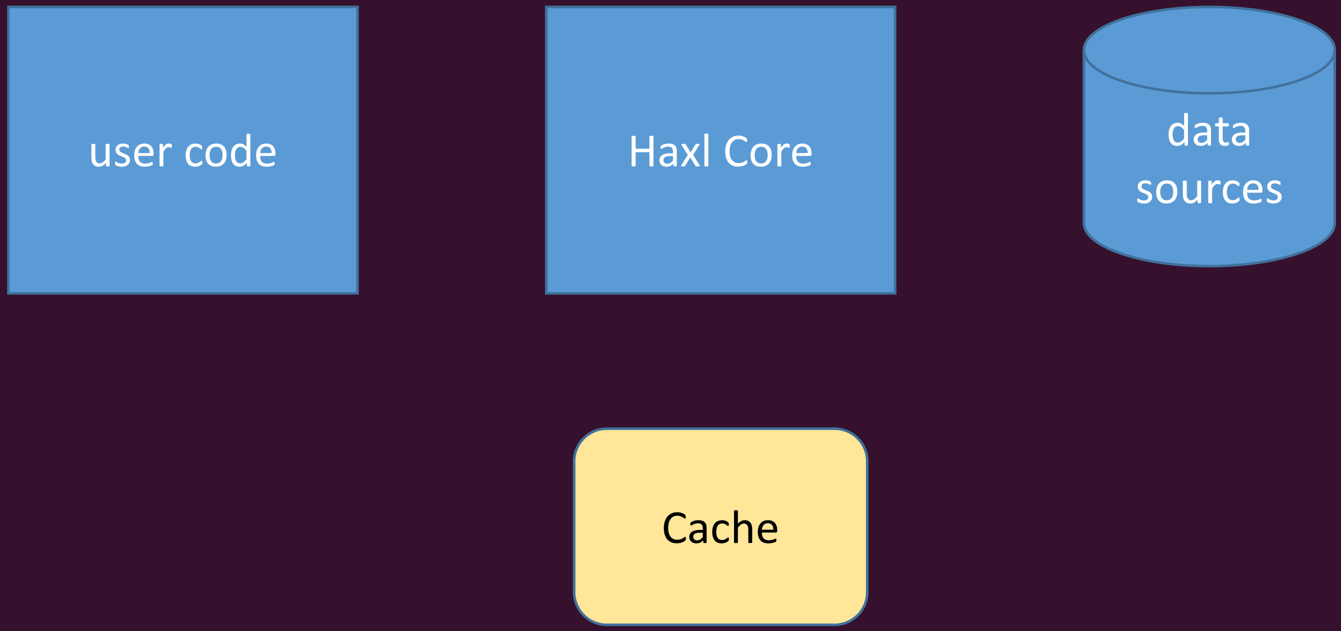
- The Haxl monad and the type system give us:
  - Guarantee of no side effects, except via dataFetch
  - Guarantee that everything is cached
  - The ability to replay requests...

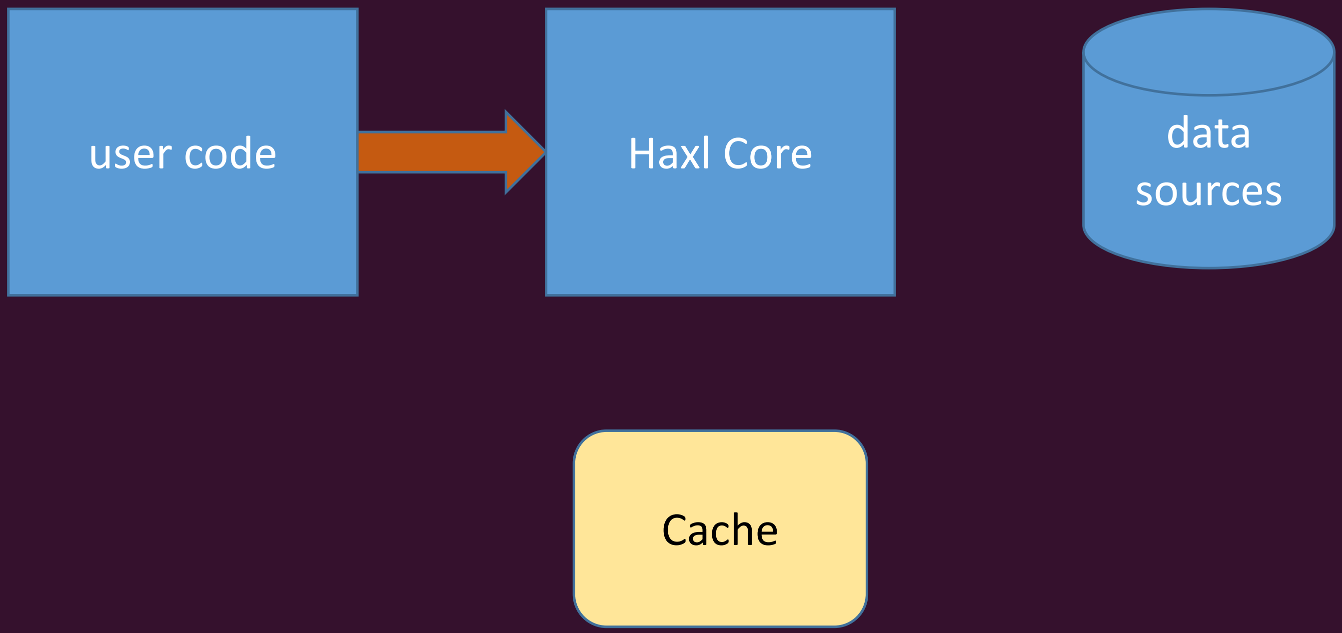
user code

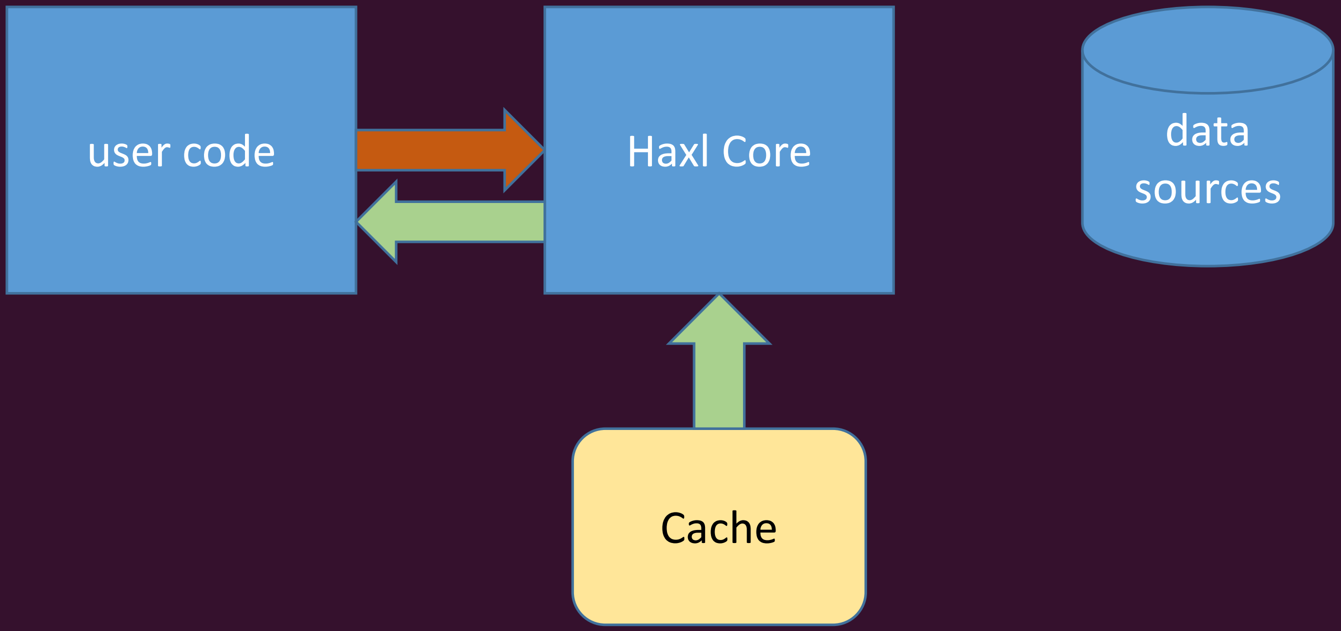

user code
Haxl Core

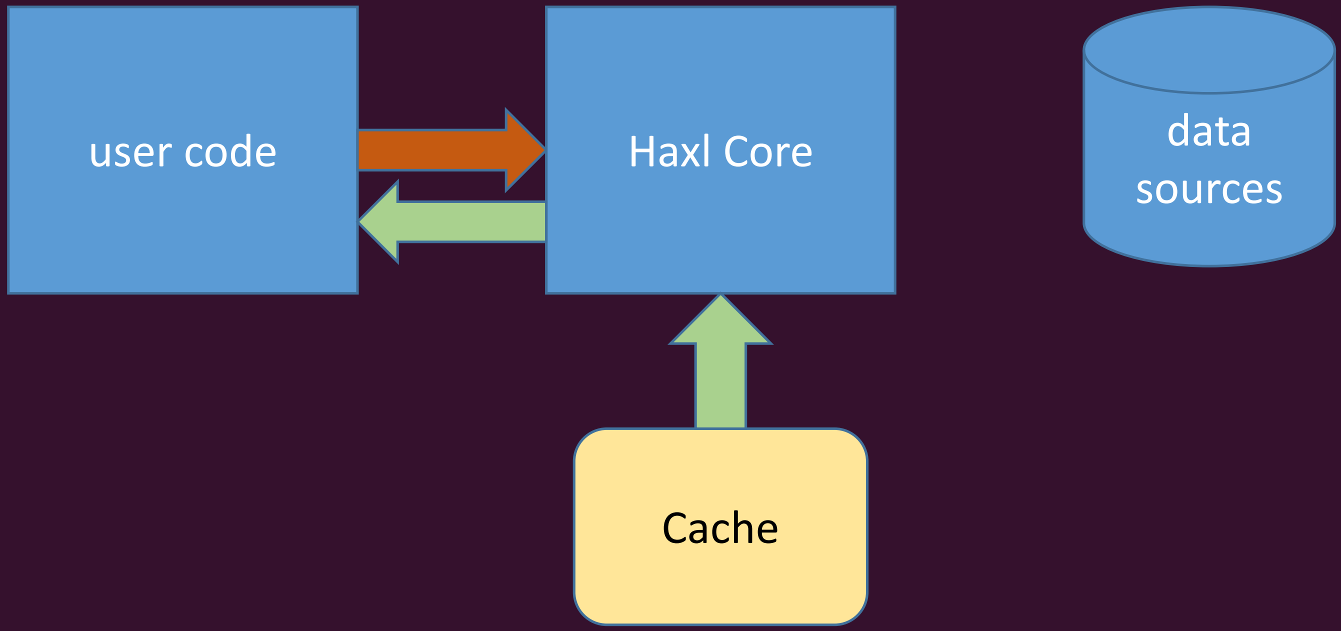

user code
Haxl Core
data
sources


user code
Haxl Core
data sources
Cache


user code
Haxl Core
data
sources
Cache


user code
Haxl Core
data sources
Cache


user code
Haxl Core
data sources
Cache



- The data sources change over time
- But if we *persist the cache*, we can re-run the user code and get the same results
- Great for
    - testing
    - fault diagnosis
    - profiling

Scaling

- Each server has lots of cores, pounded by requests from other boxes constantly.

Hot code swapping

- 1-2K machines, new code pushed many times per day
- Use GHC’s built-in linker
  - Had to modify it to unload code
  - GC detects when it is safe to release old code
- We can swap in new code while requests are still running on the old code

Status

- Prototyped most features (including hot code swapping & scaling)
- Core is written
- We have a few data sources, more in the works
- Busy hooking it up to the infrastructure
- Can play around with the system in GHCi, including data sources

Questions?