Handy Documentation

Release 0.5-SNAPSHOT

William Billingsley

Sep 12, 2017

Contents

Handy is a small set of libraries that make writing asynchronous reactive apps concise, easy, and functional.

The core of handy is Ref, a monad for the web.

It's especially useful if you are writing the server for a "single page app", such as an Angular.js app, backed by Play and ReactiveMongo, but it is tidily componentised to be useful in other contexts.

CHAPTER 1

Ref is a Monad for the Web

(If you want a little primer on monads, and why they are useful, I'll add one shortly.)

Monads allow us to wrap functionality around data. This helps us to separate concerns in a generic and typesafe manner.

They also help us chain actions together using `flatMap` (or here, using Scala's for notation), producing an algorithm that reads as simply as if it were imperative code:

```
for {
  page <- LazyId(pId).of[Page]
  approved <- request.approval ask editPage(page.itself)
  updated <- update(page, json)
  saved <- pageDAO.save(updated)
} yield saved
```

But it's better than imperative code. The result of the code above would be a `Ref[Page]`. That means the functionality we've wrapped around it – such as error handling and asynchronicity – is present in the result.

What does Ref handle

- Lazy ID references

  Most applications have some kind of database, with items referenced by an ID within it. Some are synchronous; some are asynchronous. We'd like to be able to express algorithms on things whether we've just got their ID, whether they are coming from another query, or whether we've already fetched the item.

  And of course items can reference other items. A page can be in a course, written by an author. If we need their details, we want to fetch them. But if all we want from the reference is the ID itself, we don't want to go and fetch the object from the database if we can help it.

  ```
  val user = LazyId(123).of[User]
  ```

- Asynchronicity

If we're fetching data from remote systems (including the database), we may wish to do so asynchonrously rather than block the thread. `Ref` supports `Future` to allow this.

- Absence

  If we look something up by its ID, perhaps there is nothing for that ID.

- Failure

  Perhaps the database is down? Perhaps the user is trying to do something they are not allowed to?

- Plurality

  Usually applications treat one-of-something differently than many (zero-or-more) of something.

  For example, we may want to just return one item from an HTTP request (and give a 404 if there isn't one). But we may want to *stream* items from a request for it's children, and it's only a 404 if the parent doesn't exist (not if the parent has no children).

  And we might want the code to give us those plural children to be as simple as `refParent flatMap _.children`

- Enumerators (for feeding iteratees), in handy-play

  This is especially useful if you're working with an asynchronous database driver, such as ReactiveMongo, that can stream data out asynchonously.

Seamless asynchronous security

`Approval` is a concise class for doing asynchronous security. Security checks can do lookups if they need to. They can delegate to other approvals if find they need to. And permissions are remembered within an approval appropriately so that it all works very efficiently.

For example, consider editing a wiki page in a teaching course.

- We want to check for the "edit page" permission on that page.
- If the page is protected, then we need the "edit protected" permission on the course, which requires the user to be registered for the course with the "Moderator" role
- If it's not protected, we just need the "edit unprotected" permission on the course, which requires the user to be registered for the course with the "Member" role.

A typical call inside some operation might look like:

```
for {
  approved <- approval ask editPage(p)
  ... // do stuff monadically
} yield ... // whatever we want to return
```

This flow might need to look up the page, the user, and the course to work out the answer. But we'd also like to remember everything we've already been granted so we don't have to look anything up twice.

`Approval` makes checks like this incredibly concise, readable, asynchronous, and efficient.

Seamless asynchronous JSON conversion

Just as security might need to look something up, so might converting something into JSON.

For instance suppose you want to send a page to the client, but you want to send some permissions with it (so the client knows whether to enable or disable the edit button in the GUI):

```
def toJsonFor(page:Page, approval:Approval[User]) = {
  for {
    read <- approval askBoolean read(page)
    edit <- approval askBoolean edit(page)
  } yield Json.obj(
    "id" -> page.id,
    "text" -> page.text,
    permissions -> Json.obj(
      "read" -> read,
      "edit" -> edit
    )
  )
}
```

Concise support for single-page-apps

Where handy really shines is in writing the server side of single page apps.

Security, JSON conversion, asynchronicity, and detecting whether this is a JSON request for data or an HTML request (hitting refresh in the browser) so you can instead send the single page app's HTML, all wrapped up as simply as:

```
def editPage(pageId:String) = DataAction.returning.one(parse.json) { implicit request␣
↪->
  for {
    page <- LazyId(pageId).of[Page]
    approved <- request.approval ask Permissions.editPage(page.itself)
    updated <- update(page, request.body)
    saved <- pageDAO.save(updated)
  } yield saved
}
```

Note: `page.itself` is syntactic sugar for `RefItself(page)`

Streaming all its revisions using HTTP1.1 chunked?

```
def revisions(pageId:String) = DataAction.returning.many { implicit request =>
  for {
    page <- LazyId(pageId).of[Page]
    approved <- request.approval ask Permissions.readPage(page.itself)
    revision <- page.revisions
  } yield revision
}
```

Ties into EventRoom

Eventroom is an everso-simple library for apps to do publish and subscribe over websockets or server-sent-events.

Ties into handy-play-oauth

Handy-play-oauth is an everso-simple library for doing social media logins (eg, sign in with GitHub).

Detailed documentation

The documentation pages below are being expanded with explanations of how everything works and fits neatly together.

Contents:

Lazy Ids

Unsurprisingly, a `LazyId` is a referernce to something by its Id. It has two parts – a key, and how to look the item up:

```
LazyId(id).of(lookupMethod)
```

Why is it split into two calls like that? It lets us take better advantage of Scala's implicits, if we have an implicit lookup method in scope. For instance, we could write:

```
LazyId(id).of[User]
```

And if there is an implicit lookup method for Users (and that kind of key) in scope, then it will be used.

Equality for Lazy IDs

Two ‘‘LazyId‘‘s are equal if they have the same key and the same lookup method. This typically means that:

```
LazyId(1).of[User] != LazyId(1).of[Course]
```

LazyIds are lazy Refs that memoise their result

A `LazyId[User]` is a subtype of `Ref[User]`. If we call `map`, `flatMap`, or many of the other functions on the LazyId, it will call the lookup method. This uses a `lazy val`, so that once the item has been looked up, it will be remembered.

For instance, if your lookup method is asynchronous, and returns a `Future` (wrapped as a `RefFuture`), then that future will be memoised in the lazy val.

If you don't want the memoised result (for instance you have made an update that has changed the value in the database), you can always use either `lazyId.copy`, which will produce a duplicate lazy id with the same key and lookup method but with the lazy val not memoised yet. Or you could use `RefById`, which is like `LazyId` but without the lazy val (ie, no memoisation of the result).

Look up methods return another Ref

A lookup method for a single item:

```
trait LookUpOne[T, K] {
  def lookUpOne[KK <: K](r:RefById[T]):Ref[T]
}
```

It takes the key, and can return any kind of `Ref`. So, it could work asynchronously, returning a `RefFuture[T]`. Or it could work synchronously, returning `RefItself[T]`, `RefFailed`, or `RefNone`.

As the lookups for different types of item can be independent of one another, it is simple to write applications where different items live in different databases.

Getting the key

`LazyId[T]` is also an `ImmediateId[T]`. This means that the ID for this Ref, if there is one, is immediately available:

```
val id = lazyId.getId
```

If you just use string IDs everywhere (and your data uses the trait `HasStringId`) you won't need to worry about this next bit, but we also have a mechanism to "canonicalise" ids.

Should `LazyId("1").of[User]`, `LazyId(1).of[User]` and `LazyId(1L).of[User]` all refer to the same item?

Or, if you're using something like MongoDB, what about converting strings to an `ObjectId`?

And if we have the item itself (a `RefItself`), then how do we extract the id from it? Is it in `id`, or (as in MongoDB objects) in `_id`, or somewhere else?

`LazyId.getId` takes an implicit parameter of type `GetsId[T, KK]`. This is an object that knows how to get the ID of that kind of item, and knows how to convert ids that might be in the wrong type into a "canonical" form.

Getting the key of a Future lookup

Suppose we want the ID of a reference, but we don't have a `LazyId`, we've got a database query:

```
val rUser:Ref[User] = UserDAO.byName("Algernon Moncrieff")
```

And suppose our database is asynchronous, and gives us a `Future` (wrapped as a `RefFuture`).

Clearly for this reference, we can't get the ID immediately – we'd need to wait for the database to have finished running the query, ie for the `Future` to complete.

So instead we would use the `refId` method that is defined for all ‘‘Ref‘‘s:

```
val rId = rUser.refId
```

This will give us a `Ref[K]` where `K` is the type of ID this item has.

If we want to wait (block) and get that as an `Option`, we can:

```
val blockingId = rId.fetch.toOption
```

but we can also keep working asynchronously, because `Ref` is a monad that is happy with asynchronicity:

```
for {
  id <- rId
} yield ...
```

Generally doing the latter (using the monadic `map` and `flatMap` methods, or `for` notation) is better. If the ID is immediately available, it will run immediately (negligible cost); if it is a Future computation it will run when the Future is complete.

Lookup caches

Although `LazyId``s memoise their results, sometimes you want to ensure that even if you have another ``LazyId` to the same item, it won't repeat the lookup.

This is what `LookupCache` is for. The class is available to you; it's up to you when you use it.

Remember, two `LazyId``s are equal if their IDs and their lookup methods are the same. So, a ``LookupCache` keeps a simple concurrent mutable map of `LazyId` to looked up `Ref`.

A typical usage might be:

```
for {
  item <- cache.lookUp(lazyId)
} ...
```

This will return the cached `Ref` if it is available, or cache the `LazyId` (which memoises its result) if there is not already a value in the cache.

If you have an item, you can also pre-seed a cache with it:

```
val item = User(id=1, name="Algernon Moncrieff)
cache.remember[Int, User](item.itself)
```

This uses an implicit lookup method and an implicit `GetsId`. The reason being that we need to construct a `LazyId` of the canonical type, including the lookup method it would use, and then store this `Ref` as the cached value for it.

Lookup catalogs

`LazyId(1).of[User]` requires a look up method to be passed in (implicitly) at compile time.

It may be that you'd rather delay that until runtime – keep a registry or catalog of lookup methods, and have your start-up configuration register lookup methods that will be used.

This is what `LookUpCatalog` is for:

```
val catalog = new LookUpCatalog
val ref = catalog.lazyId(classOf[User], 1)
```

And separately:

```
catalog.registerLookUp(classOf[User], userLookup)
```

You'll notice that when using `LookUpCatalog`, we pass in a `Class` object when we want to register a look up method, or create a `LazyId` that uses the catalog. This is because of type erasure in Scala (and Java).

Two `LazyId``s from a ``LookUpCatalog` are considered equal if they have the same ID, the same class (passed in), and come from the same catalog.

Approval

`Approval` is a concise and simple way of doing asynchronous security, that lets you express permission rules of arbitrary complexity. Security checks can be asynchronous (for instance if they want to make an asynchronous database lookup). They can delegate to other approvals if find they need to. And permissions are remembered within an approval appropriately so that it all works very efficiently.

There are four very simple types to understand:

- `Approved` is a simple token just to say something has been approved.
- A permission is an object of type `Perm` that has a single method, `resolve`, which will decide (possibly asynchronously) whether to approve or refuse the permission. It returns a `Ref[Approved]`. The permission is approved if that results in an approval (for example `RefItself(Approved)` or a `RefFuture` that is successful).
- `Approval` collects these permissions when they are asked for and approved. It has the `ask` method that is used to ask for a permission. It has also holds a `Ref[U]` to the user the permissions are being collected for, in the value `who`.
- `Refused` is a failure that says the permission has been refused. If a permission returns `Refused(...)` it is implicitly promoted to `RefFailed(Refused(...))`.

Unique permissions

As a simple first example, let's consider designing a "create course" permission where the user can create the course if they have the role "Teacher" on the site:

```
val createCourse = Perm.unique[User] { (prior) =>
  (
    for {
      user <- prior.who if user.hasRole("Teacher")
    } yield Approved("Yes, you are an author")
  ) orIfNone Refused("You need to have the Teacher role to create courses")
}
```

What's going on here?

`Perm.unique` defines a new permission. You could also do this by directly subclassing `Perm`. The resolution method is passed the `Approval` object, called "prior" because it caches all the prior approvals that have been granted on it.

`prior.who` is a `Ref[User]` – a reference to the user being approved. Its value is set when the `Approval` object is created. Typically it's likely to be a `RefFuture` as most applications resolve which user is making an HTTP request by taking the session cookie and querying a database (asynchronously) to see which user has that session open.

The logic in the permission filters `prior.who`, requiring the "Author" role. This will result in the user if they are an author, or none if they are not.

As our permission check is likely to be a step in an algorithm, we'd like the result to have a nice failure message rather than just producing none, so we write `orIfNone Refused("...")` to change it from none to a failure.

Using this permission in an algorithm

The permission we've defined now fits neatly into monadic algorithms. For example, our algorithm to create a course might be:

```
for {
  approved <- request.approval ask createCourse
  created  <- pageModel.newCourse(json)
  saved    <- db.saveNew(created)
  json     <- toJson(saved)
} yield Ok(json)
```

And if the user making the request is not a teacher, the result will be `RefFailed(Refused("You need to have the Teacher role to create courses"))`

Equal on ID

Most permissions in a system are about an item. For example, we might want the user to be able to edit a page within a course. This might be:

```
val editPage = Perm.cacheOnId[User, Page] { case (prior, refPage) =>
  ...
}
```

This gives us `editPage`, that can produce permissions asking to edit particular pages. So, `editPage(page123.itself)` and `editPage(LazyId(123).of[Page])` are permissions asking to edit page 123.

Why `Perm.cacheOnId`? If the user is approved to `editPage(page123.itself)`, then a later request for approval to `editPage(LazyId(123).of[Page])` will find the previously granted permission because both permissions come from `editPage` and refer to items with the same ID.

Delegating permissions

On many occasions, permissions will want to delegate to other permissions.

For example, suppose there are "protected" and "unprotected" pages in a course, and students are allowed to edit the unprotected pages, but only moderators can edit the protected ones. We could simply write this as:

```
val editPage = Perm.cacheOnId[User, Page]({ case (prior, refPage) =>
  for {
    page <- refPage
    approved <- {
      if (page.protected) {
        prior ask editProtected(page.course)
      } else {
        prior ask editUnprotected(page.course)
      }
  } yield approved
})
```

Summing up

Approval lets us express our security rules in terms of our application classes.

It's independent of the web framework we choose to use, and independent of the datastores we plug in to look up our data.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search