Certificate Ocap Ledger HOWTO

Version 7.0

Christopher Lemmer Webber

September 18, 2018

This tutorial introduces the basics of how to construct a "ledger" where objects can be inserted into the ledger and access control is enforced through certificate-style object capabilities. It is assumed that anyone can view the ledger's chain of events, but only entities on the ledger itself can authorize access to themselves.

Object capabilities are an authority mechanism based on authority-by-possession rather than authority-by-identity. Object capabilities can be implemented on a variety of "substrates". In the interest of ease of development and experimentation, this code was written in a mashup of certificate-style object capability system (such as ocap-ld) but implemented on top of a substrate that uses local encapsulation for its integrity. Sorry... in the near future maybe this will be updated to just use ocap-ld.

Originally this document included an introduction to object capabilities as well but the author decided this thing was already way too long. Sorry, the two documents linked in the previous paragraphs contain decent introductions so look at those instead.

1 Just enough Racket to follow along

This document uses the Racket programming language (mainly because the relevant code was written in the author's off hours to see if all the ideas worked). [1] Racket may look a bit different than other programming languages you've seen, but it's frequently used to teach middle schoolers how to program... which means you can probably learn it quickly, too.

It's recommended that you check out the associated repository and play around but obviously not required. If you open a "test.rkt" file in DrRacket in the checkout's directory and hit "run", you can then enter all the examples in the REPL (and put any data you want to keep in the top part of the program).

Only a few things need to be said about Racket's syntax. Instead of calling a function like this:

```
stringAppend("foo", "bar") ;; => "foobar"
```

We call it like this:

```
> (string-append "foo" "bar")
"foobar"
```

These are the main core types we'll be using:

```
42                          ; numbers
(list 1 2 3)                ; linked lists
'(1 2 3)                    ; same as above but quoted
"foo"                       ; strings
'foo                        ; symbols
#hasheq((name   . "bob")    ; immutable hash tables
        (drinks . "tea"))
#t #f                       ; booleans (true/false)
```

Defining a variable uses `define`:

```
> (define my-name
    "Alice")
> my-name
"Alice"
```

Defining and then invoking a function also uses `define`, but notice how the function name is wrapped in parentheses:

[1] This is not a full tutorial on the Racket programming language; if you are interested in that, the Racket documentation is very good, and this tutorial is a fun way to get your feet wet.

```
> (define (greet name)
    (string-append "Hello " name "!"))
> (greet my-name)
"Hello Alice!"
```

Some functions can return multiple values, which we can capture with `define-values`:

```
> (define (an-animal-and-noise)
    (values 'pig 'oink))
> (define-values (animal noise)
    (an-animal-and-noise))
> animal
'pig
> noise
'oink
```

Finally, in the long history of lisps there is an idea that code and data are not so far apart. As demonstration of this, we can "quote" an expression, and we'll be able to reference that expression as data instead of code:

```
> (+ 2 2)
4
> '(+ 2 2)
'(+ 2 2)
```

Finally there's a language feature called "quasiquoting" which uses a backtick instead of an apostrophe. Quasiquoting is really a templating language of sorts, which allows us to switch between code and data very easily by "unquoting" with a comma.

```
> `(evaluating (+ 2 2) gives us ,(+ 2 2))
'(evaluating (+ 2 2) gives us 4)
```

This is especially frequently combined with the hash table syntax, mentioned above.

Okay, that's all you need to know for Racket syntax for this tutorial!

2 Let's get into it

First of all, we'll want to import a few things:

```
> (require "ocap-certs.rkt" "ledger.rkt")
```

2.1 Creating the root of the ledger

On our ledger chain, our ledger itself will have representation as a slightly-special document. Like everything else, it needs a way to delegate capabilities related to itself, so it'll need a keypair too.

```
> (define-values (ledger-privkey ledger-pubkey)
    (new-key-pair 'ledger))
> (define ledger-doc
    `#hasheq((type . (ledger))
             (name . "ledger root")
             (delegate-key . ,ledger-pubkey)))
```

Now to create the chain. Our chain is going to be a linked list, for simplicity's sake. [2] Every ledger needs to start somewhere, and our ledger needs a genesis block. Here's ours:

```
> (define minimalist-chain
    (list (genesis ledger-doc basic-ledger-actions basic-objekt-
actions)))
```

The root object on our chain is wrapped in a special `genesis` structure which holds the document that identifies the ledger itself, actions (ie "methods") that can be used to update the ledger, and actions that can be used so that objects may update *themselves*. (This can be updated too over time, so we can change the expected behavior of how our ledger operates.) We'll worry about actions later.

This doesn't tell us anything about the actual state that's been generated from our ledger though... we have to process the ledger to get.

```
> (define minimal-ledger
    (process-whole-chain minimalist-chain))
> minimal-ledger
#<ledger>
```

[2] By making it a linked list, we're leaving out an important detail. While the validity each invocation on this chain is secured, we haven't enforced an ordering of events, which could lead to malicious reordering. A real ledger would have each item point at the previous "commit" on the chain, but for the simplicity of printing things to screen, we're just using a linked list. Relatedly, a distributed ledger in a mutually suspicious environment would also have to integrate consensus.

Okay, so this is a structure that represents the "state of the ledger" and the objects on it, as well as what actions are "active". We need some sort of identifier / key to fetch the latest version of an object. It would be sensible to generate the identifier based off something identifying the initial object on the chain. We could take the hash of the object's delegation key for instance, though this adds some complexity (we will look at how to do deal with that later) but a much easier option would be to simply normalize and hash the initial object version and use *that* as the key.

However, this toy ledger is just being done in memory so we will cheat and use the initial object itself as the key (the hash table we are using indexes by whether the object points to the *same* object in memory, rather than an equivalent one).

There's only one object our ledger's state so far, so we might as well extract it:

```
> (ledger-ref-doc minimal-ledger ledger-doc)
'#hasheq((delegate-key . #<pubkey ledger>)
         (name . "ledger root")
         (type . (ledger)))
```

Hm... well we haven't made any changes yet! So this is just the ledger document as it initially stood. [3]

2.2 Adding another object to the ledger

It's silly to have a ledger with only a ledger document on it. This is a toy ledger, so why not add a toy object:

```
> (define-values (spaceman-privkey spaceman-pubkey)
    (new-key-pair 'spaceman))
> (define spaceman-doc
    `#hasheq((type . (toy))
             (name . "Gus Lightwave")
             (catchphrase . "Infinity... the final frontier!")
             (delegate-key . ,spaceman-pubkey)))
```

Now we need to make an invocation against the ledger to put our spaceman *on* the ledger. We're going to cheat on this first one... an object can always invoke itself as a target. So the ledger will add this one.

```
> (define add-spaceman-capinv
    (make-invocation ledger-doc 'register-doc
```

[3] Actually there's one more bit of information we're not showing... an object can store metadata state that isn't represented on the published object but may be used to change behavior of future object invocations. Use `ledger-ref` to see the full structure.

```
                    `#hasheq((document . ,spaceman-doc))
                    ledger-privkey ledger-pubkey))
```

The first argument to `make-invocation` is the capability we're invoking... in this case, the target is always a capability to its own delegate-keys, so we're letting the ledger invoke the ledger. The second argument is the action we want to invoke... in this case, we want to register the document, so it's `'register-doc`. Next is the arguments; this action takes one arggument, the `'document` being added. Finally the private and public keys that have authority to invoke this capability.

Let's take a look at the generated capability invocation document:

```
> add-spaceman-capinv
'#hasheq((document
          .
          #hasheq((catchphrase . "Infinity... the final
frontier!")
                  (delegate-key . #<pubkey spaceman>)
                  (name . "Gus Lightwave")
                  (type . (toy))))
         (proof
          .
          #hasheq((capability
                   .
                   #hasheq((delegate-key . #<pubkey ledger>)
                           (name . "ledger root")
                           (type . (ledger))))
                  (creator . #<pubkey ledger>)
                  (proofPurpose . cap-invoke)
                  (sig . #<signature by ledger>)))
         (type . register-doc))
```

Looks about right... let's make an updated version of the ledger with this item applied:

```
> (define ledger-with-spaceman
    (ledger-update minimal-ledger add-spaceman-capinv))
> (ledger-ref-doc ledger-with-spaceman spaceman-doc)
'#hasheq((catchphrase . "Infinity... the final frontier!")
         (delegate-key . #<pubkey spaceman>)
         (name . "Gus Lightwave")
         (type . (toy)))
```

Sweet, that looks right.

2.3 Paying to get on the ledger

One problem though... currently only the ledger can add items to itself. Mr. Tomato Head thinks this looks fun and would like to join the ledger:

```
> (define-values (tomato-delegate-privkey tomato-delegate-pubkey)
    (new-key-pair 'tomato-delegate))
> (define-values (tomato-autograph-privkey tomato-autograph-
pubkey)
    (new-key-pair 'tomato-autograph))
> (define tomato-head-doc
    `#hasheq((name . "Tomato Head")
             (catchphrase . "That's MISTER Tomato Head to you!")
             (delegate-key . ,tomato-delegate-pubkey)
             (autograph-key . ,tomato-autograph-pubkey)))
```

One problem: Mr. Tomato Head doesn't have access to and doesn't know anyone with access to the ledger's private key. But what Mr. Tomato Head has is money. It turns out we can implement money using object capabilities. Money has to come from somewhere, and here's the mint that issues blox bucks:

```
> (require "money.rkt")
> (define blox-mint
    (make-mint 'blox))
```

And Mr. Tomato Head's purse is lined with cash:

```
> (define tomato-purse
    (send blox-mint make-purse 10000))
```

As it turns out there's an excellent opportunity to put that money to use. One way or another a public ledger must keep itself from being spammed and have a way of distributing the upkeep of being run. One approach is a proof-of-work type system, and for now we'll imagine we've added that. [4] Another approach is to introduce "accelerators", entities which have been granted authority to be able to put things on the ledger. Let's make one:

```
> (define-values (accelerator-privkey accelerator-pubkey)
    (new-key-pair 'accelerator))
> (define an-accelerator
    (new accelerator%
         [public-key accelerator-pubkey]
```

[4] Supporting proof of work could be done by having more flexibility in what kinds of proofs we accept than just signatures. We could allow a proof type that involves completing some proof of work, and then delegate a capability to that abstract proof of work specification.

```
          [private-key accelerator-privkey]
          [purse (send blox-mint make-purse 0)]
          [register-cost 15]
          [invoke-cost 10]
          [register-doc-cap
           (cap-delegate ledger-doc ledger-privkey ledger-pubkey
                         (list accelerator-pubkey)
                         #:caveats
                         `(#hasheq((type . action)
                                   (action . (register-doc)))))]
          [post-invocation-cap
           (cap-delegate ledger-doc ledger-privkey ledger-pubkey
                         (list accelerator-pubkey)
                         #:caveats
                         `(#hasheq((type . action)
                                   (action . (post-
invocation)))))]))
```

A few things to note from the above... our accelerator also has a purse, for one. For two, our accelerator has been delegated two object capabilities... one with the caveat that it can only register documents and the other with the caveat that it can only post invocations. (These could also be combined into one caveat listing either action within the caveat.) If there are other ledger actions, such as ones that allow you to update the ledger rules, this particular accelerator doesn't have access to them.

One interesting thing is that *the accelerator is not actually on the ledger itself and does not need to be* (though we could put them or a document containing their public keys on the ledger if we wanted to be somewhat meta-circular). This accelerator is an actor and lives Somewhere (TM) but where that is doesn't particularly matter as long as we can send it messages somehow. We could do so over HTTP or whatever other compatible protocol. But since this is all in memory, we'll just send it a message using Racket's `send` expression:

```
> (send an-accelerator get-balance)
0
```

So our accelerator friend here has capabilities to post to the ledger, but no money. Our Mr Tomato Head friend has money, but no capability to post to the ledger. Thankfully an exchange can be made.

```
> (define register-tomato-payment
    (send tomato-purse sprout))
> (send register-tomato-payment deposit 15 tomato-purse)
```

Our shiny red friend doesn't want to just hand over their entire purse and let the accelerator take whatever they want (who do we look like, credit card payment infrastructure design-

ers?), so they make a one-off purse to contain just the money necessary for the transaction. Now to send to the accelerator:

```
> (define register-tomato-cap
    (send an-accelerator buy-register-cap
          tomato-delegate-pubkey tomato-head-doc
          register-tomato-payment))
```

Success! And indeed, the payment wallet is empty while the accelerator's wallet finally has some cash in it:

```
> (send register-tomato-payment get-balance)
0
> (send an-accelerator get-balance)
15
```

The accelerator sent us back that capability, so let's look at it in a bit more detail:

```
> register-tomato-cap
'#hasheq((caveats
          .
          (#hasheq((action . (register-doc)) (type . action))
           #hasheq((field . document)
                   (type . require-value)
                   (value
                    .
                    #hasheq((autograph-key . #<pubkey tomato-
autograph>)
                            (catchphrase . "That's MISTER Tomato
Head to you!")
                            (delegate-key . #<pubkey tomato-
delegate>)
                            (name . "Tomato Head"))))))
         (invokers . (#<pubkey tomato-delegate>))
         (parent-capability
          .
          #hasheq((caveats
                   .
                   (#hasheq((action . (register-doc)) (type . ac-
tion))))
                  (invokers . (#<pubkey accelerator>))
                  (parent-capability
                   .
                   #hasheq((delegate-key . #<pubkey ledger>)
                           (name . "ledger root")
```

```
                        (type . (ledger))))
                (proof
                 .
                 #hasheq((creator . #<pubkey ledger>)
                         (proofPurpose . cap-delegate)
                         (sig . #<signature by ledger>)))))
        (proof
         .
         #hasheq((creator . #<pubkey accelerator>)
                 (proofPurpose . cap-delegate)
                 (sig . #<signature by accelerator>))))
```

Hoo, that's quite an eyeful! That's because there are actually three capability documents nested together here forming the capability chain (not to be confused with the ledger chain) our tomato friend can enact on. If we follow the parentCapability field inward we'll notice that the innermost capability is the target itself. One layer out from there we see a document where the ledger had delegated to the accelerator authority to invoke the ledger, with the caveat that invocations must apply to the `'register-doc` action. At the outermost layer we see that the accelerator then delegates to Mr. Tomato Head, but with the caveats that invocations must apply to the `'register-doc` action (a duplicate of the previous caveat, but it doesn't hurt) and that the field `'document` must have the value that is precisely the same document Tomato Head paid to register. This latter one prevents Mr. Tomato Head from reusing this invocation to post more documents. Posting the normalized hash of the expected document would be another acceptable route for a production system, but strictly speaking is an optimization.

Okay, now Tomato Head needs to make an invocation using this capability and post it to the ledger.

```
> (define register-tomato-capinv
    (make-invocation register-tomato-cap 'register-doc
                     `#hasheq((document . ,tomato-head-doc))
                     tomato-delegate-privkey tomato-delegate-
pubkey))
> (define ledger-with-tomato-head
    (ledger-update ledger-with-spaceman register-tomato-capinv))
> (ledger-ref-doc ledger-with-tomato-head tomato-head-doc)
'#hasheq((autograph-key . #<pubkey tomato-autograph>)
         (catchphrase . "That's MISTER Tomato Head to you!")
         (delegate-key . #<pubkey tomato-delegate>)
         (name . "Tomato Head"))
```

Okay, looks like it works and our Tomato friend is on the ledger!

2.4 An object updates itself on the ledger

Some time passes, and Tomato Head receives an honorary PhD for his fine acting. Tomato Head wants everyone to know about this and so he wants to make an update to his catchphrase. He constructs the following invocation:

```
> (define doctor-tomato-capinv
    (make-invocation tomato-head-doc 'update-field
                     #hasheq((field . catchphrase)
                             (value . "That's DOCTOR Tomato Head
to you!"))
                     tomato-delegate-privkey tomato-delegate-
pubkey))
```

Except Tomato Head doesn't have permission to update the ledger with this invocation. So we're going to have to – hold on to your hats – get a capability to make an invocation that embeds another invocation. [5]

We can do that by paying the accelerator again, but this time asking for a capability to post an invocation rather than one to register a document.

```
> (define update-tomato-payment
    (send tomato-purse sprout))
> (send update-tomato-payment deposit 10 tomato-purse)
> (define post-update-tomato-cap
    (send an-accelerator buy-post-invoke-cap
          tomato-delegate-pubkey doctor-tomato-capinv
          update-tomato-payment))
> (define post-update-tomato-capinv
    (make-invocation post-update-tomato-cap 'post-invocation
                     `#hasheq(
                            (invocation . ,doctor-tomato-capinv))
                     tomato-delegate-privkey tomato-delegate-
pubkey))
```

Want to see this whole thing with an invoctation that embeds an invocation? It's a doozy: [6]

```
> post-update-tomato-capinv
'#hasheq((invocation
          .
          #hasheq((field . catchphrase)
```

[5] As a side note to the ocap-ld aware, this is why framed representation of ocap invocations in ocap-ld is critical. If an invocation embeds an invocation, we need to know which one is the root. Conversion to graph soup is a lossy operation.

[6] Yo dawg I heard you like capabilities so I put a capability in your capability so you can invoke while you invoke

```
                  (proof
                   .
                   #hasheq((capability
                            .
                            #hasheq((autograph-key
                                     .
                                     #<pubkey tomato-autograph>)
                                    (catchphrase
                                     .
                                     "That's MISTER Tomato Head to
you!")
                                    (delegate-key . #<pubkey
tomato-delegate>)
                                    (name . "Tomato Head")))
                           (creator . #<pubkey tomato-delegate>)
                           (proofPurpose . cap-invoke)
                           (sig . #<signature by tomato-
delegate>)))
                  (type . update-field)
                  (value . "That's DOCTOR Tomato Head to you!")))
         (proof
          .
          #hasheq((capability
                   .
                   #hasheq((caveats
                            .
                            (#hasheq((action . (post-invocation))
                                     (type . action))
                             #hasheq((field . invocation)
                                     (type . require-value)
                                     (value
                                      .
                                      #hasheq((field . catch-
phrase)
                                              (proof
                                               .
                                               #hasheq((capability
                                                        .
                                                        #hasheq((autograph-
key
                                                                 .
                                                                 #<pubkey
tomato-autograph>)
                                                                (catchphrase
                                                                 .
                                                                 "That's
```

```
MISTER Tomato Head to you!")
                                                                (delegate-
key
                                                                 .
                                                                 #<pubkey
tomato-delegate>)
                                                                (name
                                                                 .
                                                                 "Tomato
Head")))
                                                          (creator
                                                           .
                                                           #<pubkey
tomato-delegate>)
                                                          (proofPurpose
                                                           .
                                                           cap-
invoke)
                                                          (sig
                                                           .
                                                           #<signature
by tomato-delegate>)))
                                                    (type . update-
field)
                                                    (value
                                                     .
                                                     "That's DOCTOR
Tomato Head to you!"))))))
                              (invokers . (#<pubkey tomato-
delegate>))
                              (parent-capability
                               .
                               #hasheq((caveats
                                        .
                                        (#hasheq((action . (post-
invocation))
                                                 (type . action))))
                                       (invokers . (#<pubkey acceler-
ator>))
                                       (parent-capability
                                        .
                                        #hasheq((delegate-key .
#<pubkey ledger>)
                                                (name . "ledger
root")
                                                (type . (ledger))))
```

```
                                        (proof
                                         .
                                         #hasheq((creator . #<pubkey
ledger>)
                                                 (proofPurpose . cap-
delegate)
                                                 (sig . #<signature by
ledger>)))))
                                (proof
                                 .
                                 #hasheq((creator . #<pubkey accelera-
tor>)
                                         (proofPurpose . cap-delegate)
                                         (sig . #<signature by acceler-
ator>)))))
                        (creator . #<pubkey tomato-delegate>)
                        (proofPurpose . cap-invoke)
                        (sig . #<signature by tomato-delegate>)))
                (type . post-invocation))
```

Good thing we didn't have to write out this by hand... it's nice to have functions that allow us to compose this without having to think too much about the underlying structure, and any reasonable interface should do the same.

Anyway, we can now post that to the ledger and everyone can know what Mr. Tomato's new catchphrase is:

```
> (define ledger-with-new-catchphrase
    (ledger-update ledger-with-tomato-head
                   post-update-tomato-capinv))
> (ledger-ref-doc ledger-with-new-catchphrase tomato-head-doc)
'#hasheq((autograph-key . #<pubkey tomato-autograph>)
         (catchphrase . "That's DOCTOR Tomato Head to you!")
         (delegate-key . #<pubkey tomato-delegate>)
         (name . "Tomato Head"))
```

Here we saw two distinct types of invocations, composed together:

- **To update the object:** We need to do an invocation against the object itself specifying some method and arguments that allow it to change its state (both the document shown and metadata). This was the role of the `basic-objekt-actions` behavior passed into the genesis block.
- **To update the ledger:** It's not enough to just update the object, we need to get that update/invocation on the ledger. This means we need another invocation *against the*

ledger which references the invocation *against the object* as an argument. However, the ledger needs to protect itself from being overwhelmed with updates, which is why we paid for a capability to post something to the ledger from an accelerator (though again, an abstract capability that permitted proof of work could have also worked).

2.5 Object delegates authority to be updated to another entity

As one final example, Mr. (sorry, Dr.) Tomato Head can delegate the authority to update parts of himself to another entity. Tomato Head's autographs are very valuable, but he has a tendency of losing his magic signature pen at parties. Because of this, Tomato Head and his agent have agreed that the agent should have authority to update Tomato Head's `autograph-key` (but not any other properties) just in case, and gives their agent an emergency fund in case that is necessary.

```
> (define-values (agent-privkey agent-pubkey)
    (new-key-pair 'talent-agent))
> (define agent-update-autograph-cap
    (cap-delegate tomato-head-doc
                  tomato-delegate-privkey tomato-delegate-pubkey
                  (list agent-pubkey)
                  #:caveats
                  `(#hasheq((type . action)
                            (action . (update-field)))
                    #hasheq((type . require-value)
                            (field . field)
                            (value . autograph-key)))))
> (define agent-purse
    (send tomato-purse sprout))
> (send agent-purse deposit 100 tomato-purse)
```

Now the next time Tomato Head loses his pen, his agent can quickly replace it with a new one and hand that to Tomato Head later.

```
> (define-values (tomato-autograph-privkey2 tomato-autograph-
pubkey2)
    (new-key-pair 'tomato-autograph2))
> (define new-autograph-capinv
    (make-invocation agent-update-autograph-cap 'update-field
                     `#hasheq((field . autograph-key)
                              (value . ,tomato-autograph-pubkey2))
                     agent-privkey agent-pubkey))
> (define update-autograph-payment
    (send agent-purse sprout))
> (send update-autograph-payment deposit 10 agent-purse)
```

```
> (define post-new-autograph-cap
    (send an-accelerator buy-post-invoke-cap
          agent-pubkey new-autograph-capinv
          update-autograph-payment))
> (define post-new-autograph-capinv
    (make-invocation post-new-autograph-cap 'post-invocation
                     `#hasheq(
                            (invocation . ,new-autograph-capinv))
                     agent-privkey agent-pubkey))
> (define ledger-with-new-autograph
    (ledger-update ledger-with-new-catchphrase
                   post-new-autograph-capinv))
> (ledger-ref-doc ledger-with-new-autograph tomato-head-doc)
'#hasheq((autograph-key . #<pubkey tomato-autograph2>)
         (catchphrase . "That's DOCTOR Tomato Head to you!")
         (delegate-key . #<pubkey tomato-delegate>)
         (name . "Tomato Head"))
```

Whew, it worked! Note that we did not need to define a document for the agent, and there was no requirement that the agent have a document on the ledger.

That's it for the examples. But one quick note: we have done each of these updates so far by producing a new ledger state applying the update to the last one. We could instead do this all at once, and indeed in general it is necessary to be able to audit a ledger by applying all updates successively and transforming the internal state. We could so that using the `process-whole-chain` we used at the begining:

```
> (define complete-chain
    (list post-new-autograph-capinv
          post-update-tomato-capinv
          register-tomato-capinv
          add-spaceman-capinv
          (genesis ledger-doc basic-ledger-actions basic-objekt-
actions)))
> (define complete-ledger
    (process-whole-chain complete-chain))
> (ledger-ref-doc complete-ledger ledger-doc)
'#hasheq((delegate-key . #<pubkey ledger>)
         (name . "ledger root")
         (type . (ledger)))
> (ledger-ref-doc complete-ledger spaceman-doc)
'#hasheq((catchphrase . "Infinity... the final frontier!")
         (delegate-key . #<pubkey spaceman>)
         (name . "Gus Lightwave")
         (type . (toy)))
> (ledger-ref-doc complete-ledger tomato-head-doc)
```

```
'#hasheq((autograph-key . #<pubkey tomato-autograph2>)
         (catchphrase . "That's DOCTOR Tomato Head to you!")
         (delegate-key . #<pubkey tomato-delegate>)
         (name . "Tomato Head"))
```

2.6 Key takeaways from code examples

Now that we've gone through all this, what can we learn from these code examples?

- The ledger has a genesis root document and initial set of behaviors for both updates to the ledger structure and for updates to objects.
- Each update to the ledger must be an invocation against the ledger object, using one of the specified ledger actions.[7]
- One key ledger action is the ability to add objects to the ledger.
- Another key ledger action is the ability to post invocations against objects on the ledger. These are then evaluated and applied against the object actions, allowing the object to update "itself". [8]
- Targets can delegate capabilities with the option to restrict them via caveats.
- Not all participants invoking authority on the ledger need to themselves necessarily be on the ledger.

[7] Not shown: this can include a ledger action to change the behavior of future invocations against actions for the ledger and objects.

[8] Not shown: allowing different behavior dispatched based on a combination of object + action type.

3 Using something else as the document identifier

In this document we've embedded entire documents as a way to verify same-ness. Production systems will probably want to choose a more optimal solution. Particularly, the identifier for the initial document can be canonicalized and hashed and that's an excellent way to select an identifier to refer to the object by. This is safe and highly malleable. And although other participants could upload documents referencing keys that they do not "own", it would be a waste of their resources.

Another way to do it would be to take the key fingerprint of the `delegation-key` or equivalent and use that as the id. This would require attaching another proof to the document which self-signs it with the delegation key, with a relevant proofPurpose describing the signature's intention.. This seems like a strange decision though when one could just hash the whole document.

4 TL;DR, just show me how to do it in ocap-ld

This is an attempt to sketch out the above workflow in ocap-ld.

TO BE WRITTEN SOON