Introduction

Ziniki is many things: a company, an ecosystem, a philosophy, a microservice container. For the purposes of this guide, we will try to focus on Ziniki as an ecosystem in which it is possible to write small units of client-side code - cards - which, embedded in the ecosystem, can access the services and resources of the ecosystem without reference to location or scale.

Everything about FLAS is designed with *location transparency* in mind: there is, or should be, no difference between the way in which a card interacts with a local service to how it interacts with a remote service. At the implementation level, and in reality, there are differences of course: differences in reliability, in latency, in cost. But these differences should not be things that you, as a developer, should be concerned with. As much as possible the overall Ziniki ecosystem attempts to abstract things like security and connectivity away from you. You `require` that a service is made available; so it is (or is not). Your use of it proceeds independently of whether it is available or unavailable, connected or disconnected, local or remote; determining these things is considered advanced usage.

But from time to time we will address Ziniki the microservice container. Some of the Ziniki code "leaks" to the client side, but as much as possible the code is held in a microservice container which in turn deploys a range of services - some "built in", others provided by you or third parties - to provide the cloud services you need to build leading edge applications with minimal time, effort or bugs.

And, of course, we will not be able to resist talking about Ziniki the philosophy: separation of concerns, event driven systems, modular architectures, keeping things simple, infrastructure as a service and many more.

Organization of this Guide

As with the *FLAS Developer Guide*, this guide is intended to explain how to build FLAS applications that use Ziniki services by presenting small examples of how individual challenges can be tackled.

Prerequisites

In order to use Ziniki effectively, you must use FLAS. It is assumed that you either have learnt to use FLAS already, or are learning FLAS and Ziniki together, with all the appropriate guides and samples to hand.

All the prerequisites for FLAS are prerequisites for using Ziniki.

In order to get any of the samples to work, you will either need to configure a local, small-scale in-memory/on-disk Ziniki microservice container or subscribe to a cloud-based Ziniki provider. Setting up a local server is described in the course of the "Hello, Ziniki" example.

Other Reading

FLAS Developer and Reference Guides

There should be a guide to the Expenses Demo

Ziniki Reference Manual

1 Hello, Ziniki

With a microservice container, it is not quite possible to have the first example be "Hello, World". That very simple example does not require any reference to the microservice container and has already been covered in the FLAS Developer Guide.

Consequently, even the simplest complete Ziniki example is actually quite complicated and involves - among other things - starting a microservice container, configuring it with a TLS certificate, installing the sample and accessing it. On top of that, because Ziniki is fundamentally intended to be distributed, cloud-based architecture, we have to bend the truth a little in order to get everything working locally.

On the upside, once we have done all these things to get a simple example working, the samples in subsequent chapters will flow very much more easily. This really is a case of our infrastructure making "simple" things seem complicated, while complicated (and even complex) things become relatively simple.

About the simplest thing that we can do with Ziniki is to store a personal (to a specific user) counter on Ziniki and have it greet you the first time you start the application, and then remind you on subsequent occasions of how many times you have been greeted in the past.

1.1 Our First Service

When developing FLAS applications, we encountered a couple of client-side services (`Repeater` and `Ajax`). Now we are going to meet our first *server-side* service, the *data* service. The data service provides functionality analogous to a database (in particular, a document database such as ObjectStore, Couchbase or DynamoDB) in which you can start from a well-known "root" document and then traverse the data store by following links. We will tackle the increasing complexity of this in subsequent chapters, but for now we will just consider the simple task of getting a single, private entry.

```
entity AccessCount
    Number accesses
```

```
    card Hello
        state
            String msg <- ""

        template ziniki-hello
            message <- msg

        requires org.ziniki.DataStore ds

        implements Lifecycle
            ready
                <- ds.my (type AccessCount) -> TrackAccesses

        handler org.ziniki.DataHandler TrackAccesses
            missing
                <- ds.putMy (type AccessCount) (AccessCount
                                1)

We only want to greet the user on this client once, so
                                check if the greeting has
                                already been completed
before issuing a greeting

            value v
                | msg == ""
                    msg <- display
                    count.accesses <- count.accesses +
                                1
                    <- ds.put count
                |
                    <- Debug "we have already been greeted
                                on this client"
                        count = cast AccessCount v
                        display
                            | count.accesses == 1  = "Hello
                              From Ziniki"
                            |                      = "Ziniki
                              accessed " ++ (show count.accesses)
                              ++ " times"
```


At first sight, this may look intimidating - particularly for a "hello, world" program. While this is true, it is also the case that the simplest Ziniki program involves a number of moving parts, so it is more than just printing hello world - we need to store and retrieve records from the server.

First off, we declare an `entity` which is the object we are going to store on the server. An `entity` is essentially the same as a `struct`, except that when stored on the server it is given a unique identity which means that we can make stronger statements about whether two entities represent the same thing than we can with `structs` - where it is simply a question of whether they have the same value. This enables us to happily distribute the `entity` and be able to describe it, and update it, uniquely.

```
    entity AccessCount
        Number accesses
```

In this case, the entity is there simply to record the number of times the page has been accessed and so contains a single number with that value in it.

The basic card consists of a state containing the message we want to display and a template which binds that into the card template.

```
    card Hello
        state
            String msg <- ""

        template ziniki-hello
            message <- msg
```

Then we move on to the Ziniki specific code. First, we declare that we want to use the `DataStore`

```
    card Hello
        ...
        requires org.ziniki.DataStore ds
```

This `requires` directive says that this card depends on a given service (`org.ziniki.DataStore`) and wants to be able to use it by referencing the variable `ds`. The name of the service must match a service which is provided; the variable name is an arbitrary choice from the programmer within the scope of the card.

```
card Hello
    ...
    implements Lifecycle
        ready
            <- ds.my (type AccessCount) -> TrackAccesses
```

The `Lifecycle` contract is a standard FLAS feature which enables a card to react to events during its creation and release.The `ready` method is called when the card has been fully set up and all the contract variables (i.e. `ds`) have been initialized and are ready to use.

In this case, we send a message to the data store asking for the current user's copy of the `AccessCount` object and to have the current value (and any updates) sent to the `TrackAccesses` handler.

The `TrackAccesses` handler follows the pattern of the `org.ziniki.DataHandler` handler contract and has two cases: one for when the object is not found (`missing`) and one for when the value is found or is updated (`value`).

In the `missing` case, we know that we have never seen this object before, so we create a new one, setting the access count to 0.

```
card Hello
    ...
    handler org.ziniki.DataHandler TrackAccesses
        missing
            <- ds.putMy (type AccessCount) (AccessCount
                        1)

We only want to greet the user on this client once, so
                            check if the greeting has
                            already been completed
before issuing a greeting
```


Finally, we need to consider what happens when we receive a valid update. This actually covers three cases:

- When the object exists and is provided to us in response to the `my` message request;
- When the object does not exist, but is created by `missing`;
- When the object is updated (by us or anyone else).

For reasons discussed in the commentary, on this occasion we want to avoid the third case, so we check before doing anything else that we have not been through this code before by checking the value of the `msg` variable. This is initialized above to be the empty string (`""`), so we only take any action if it still has this value; we will update the `msg` variable during processing, so after that it will after a proper message value.

```
card Hello
    ...
    handler org.ziniki.DataHandler TrackAccesses
        ...
        value v
            | msg == ""
                msg <- display
                count.accesses <- count.accesses +
                          1
                <- ds.put count
            |
                <- Debug "we have already been greeted
                          on this client"
                    count = cast AccessCount v
                    display
                        | count.accesses == 1  = "Hello
                          From Ziniki"
                        |                      = "Ziniki
                          accessed " ++ (show count.acce
                          ++ " times"
```

1.2 Unit Tests

Much of the code that you will write, both in simple FLAS and Ziniki, cannot be tested effectively using unit tests: the tests just end up being duplicates of the code you have written. On top of that, FLAS generates much of the boilerplate code that you need, relieving you of needing to write tests at all.

But from time to time, portions of the code - particularly handler portions - become complex enough that it is worth checking what they do. The `value` handler here is a case in point. The `missing` handler simply makes a request; we could check that it does, indeed, make a request if we wanted, but it is hard to know what we would be testing.

With the `value` handler on the other hand, there are a number of cases we need to consider, based on the initial value of `msg` and the value of the `AccessCount` object passed in. So there are three unit tests that we have written to handle this.

```
test that we say hello if this is the first time we
                          receive a message
    data Hello hello
    data org.ziniki.DataHandler callback <- TrackAccesses
                          hello
    contract callback org.ziniki.DataHandler value
                          (AccessCount 1)
    assert (hello.msg)
        "Hello From Ziniki"
```

This test first creates a `Hello` card, and then - because it is a unit test - manually creates an instance of the `TrackAccesses` handler. Because this handler exists *inside* the card - and thus has access to its state - it is necessary to pass the constructor the card instance that it should reference (this happens automatically in real code because of nesting, but obviously that can't happen in a unit test). The handler is stored in a `data` value with the contract type, because that is the only way in which it can be referenced.

The handler is then called using the `contract` test construct. In this case, because it is a handler, the contract name is redundant but must still be supplied for compatibility reasons. It is the `value` method we want to call, and we pass in an `AccessCount` entity with the count set to 1, which is what will happen in response to `missing` being called.

Finally, we assert that the message in the state is what we were expecting. We could also check that this turns up where we expect it in the resulting HTML, but - to me, at least - this is redundant and duplicative on this occasion.

```
test that we provide a count if we have have been
                          greeted before
    data org.ziniki.DataStore store
    data Hello hello
    data org.ziniki.DataHandler callback <- TrackAccesse
                          hello
    expect store put (AccessCount 3)
    contract callback org.ziniki.DataHandler value
                          (AccessCount 2)
    assert (hello.msg)
        "Ziniki accessed 2 times"
```

This test is essentially similar to the first. The big difference, of course, is that we pass in an `AccessCount` entity with a value of 2, which changes the behavior. More simply, this changes the message that it is displayed, which is checked at the end.

But in this path through, we also update the entity in the data store using `ds.put`. To do this, we need to check that this message is sent with the correct (updated) value of `AccessCount`. To do this, we need to provide a mock instance of the data store. By creating a `data` item with the `org.ziniki.DataStore` contract, this is automatically wired into the card when the card is created. Then, before we call the contract, we add a deliberate `expect` that the `put` method will be called on this with an `AccessCount` object set to 3.

```
test that nothing happens if the greeting has already
                          been displayed
    data org.ziniki.DataStore store
    data Hello hello
    shove hello.msg
        "We have been greeted"
```

```
data org.ziniki.DataHandler callback <- TrackAccesses
                        hello
contract callback org.ziniki.DataHandler value
                        (AccessCount 2)
assert (hello.msg)
    "We have been greeted"
```

The final test handles the case where this specific instance of the card has already had its greeting. There are two ways of constructing this test: one would be to go through the logic twice, thus ensuring that the first call set up the correct conditions for the second call; the other (which we have followed) is to directly set the `msg` field on the card so that it is in the expected state.

This test first creates the mock data contract and the card, then "shoves" a message into the card's `msg` field. It then creates the handler `callback` and invokes it in exactly the same way as we did in the second test. Finally, it checks that the message is unchanged after this call.

Note that by creating the mock contract - even though we set no expectations on it - we will cause the test to fail if any methods are called on it. The default behavior ignores and suppresses all calls.

1.3 System Tests

Compared to Unit Tests, System Tests enable us to work at a much higher level and confirm that all the components work together, repeatably and automatically but without having to set up complex infrastructure or deal with erratic failures. We also gain a certain amount more control over our components than we would have if they were located inside remote containers.

Again, unlike Unit Tests, which attempt to be independent and isolated from the bigger picture, a System Test unfolds as a *story*. Each test builds on the previous tests and, once one test has failed, the remaining tests are abandoned.

System Tests are part of the core FLAS offering, but they come into their own in more complex environments such as Ziniki. Ziniki extends the System Test offering by allowing the tests to configure in-memory Ziniki servers with pre-canned data that evolves (repeatably) as the test is run.

The `configure` step of the system test attempts to set up all the things we need - in this case a Ziniki instance called `my.com` and a client with a `Hello` card. The `configure` step may fail, but it should not have any assertions - it is then to put in place the pieces we need in order to run the test.

```
configure
    ziniki "my.com"
        datastore "datafiles"
        user "gareth"
    data Hello hello
```

The `ziniki` command within a system test indicates that there is a Ziniki Server operating on the indicated domain (the argument to the command, in this case `my.com`). The indented block are commands to be directed at this specific server.

In order to provide the data service, it is necessary to specify the `datastore` subcommand. This takes an argument which is the directory in which to look for initial configuration files for the data store. This will be discussed in detail in later chapters. If the directory does not exist - or is empty - then the data store starts off essentially uninitialized, with only the minimal amount of scaffolding elements present (such as domains and users).

The `user` subcommand indicates that the given user must exist and that they are the "current" user in the scope of the test. Furthermore, it connects the client environment (where cards will be created) to the Ziniki environment as if this user had logged on, thus saving the tests from an excessive burden of boilerplate code.

Then we create a new `Hello` card using the `data` command.

Now we can move on to actually testing the card works in the way we expect. Because Ziniki has been initialized clean, we can be sure that there are no `AccessCount` objects in the data store.

```
test we are greeted the first time
    contract hello Lifecycle ready
    assert (hello.msg)
        "Hello From Ziniki"
```

Our first step is to call the `Lifecycle.ready` method on the card. This invokes a full cycle of creation. Internally, the system test tracks all activity on the card, in Ziniki and communications between the two and this command does not complete until the whole system is quiescent. If it takes too long to become quiescent, the test will fail with a timeout message.

Once the `ready` process has completed, the message is checked to ensure that it has the expected value.

Using the Ziniki command, it is possible to prod inside the internal state of the Ziniki data store. This is not "easy" or intuitive, and should probably be avoided in general in favor of testing things that show up in external behaviors. However, with a sufficiently complex system, it will be simpler to directly interrogate Ziniki than to test every possible behavior. The second test step does exactly this.

```
test a record was created in Ziniki
    ziniki "my.com"
        bind org.ziniki.MyRecord my "my://my.com/org.zinapps.samples.ziniki.hello.AccessCount/test302"
        bind AccessCount curr my.pointsTo
    assert (curr.accesses)
        2
```

This again uses the `ziniki` command, identifying the server serving the `my.com` domain. From here, we can use the `bind` subcommand to extract individual data elements from the Ziniki data store and introduce them into the test scope for later evaluation.

The first bind command finds the root reference record for the `AccessCount` type for the current user. Surprising as it may seem that "test302" on the end of the URI is the user identity. The "my" record is not actually the entity itself (the `AccessCount`) but is a reference to it, so the second `bind` command is needed to retrieve the actual record by resolving the `pointsTo` field of the `MyRecord`.

The `bind` command takes three arguments: the type of the entity to be recovered; a name to give to the recovered entity; and an expression (constants expressed as `String` or `URI` objects) which can be resolved to the URI of a database entry.

The third step attempts to demonstrate that nothing happens if the handler is called a second time with the same value.

```
    test nothing happens if we update the record
        contract hello Lifecycle ready
        assert (hello.msg)
            "Hello From Ziniki"
```

This asserts that even after the handler is called with a putative update, the message does not change. This test is something of a cheat, since in reality the `Lifecycle.ready` method will never be called multiple times on a card, but it is not possible to intercept the handler in a system test in the same way that we did it on the unit test.

```
    test a second client gets a different message
        data Hello second
        contract second Lifecycle ready
        assert (second.msg)
            "Ziniki accessed 2 times"
        assert (hello.msg)
            "Hello From Ziniki"
```

Finally, we start up a second `Hello` client and demonstrate that it has the messages `Ziniki accessed 2 times` while the first card's message remains unchanged.

1.4 Installing Ziniki

Ziniki builds on top of FLAS and so before installing the Ziniki runtime, you need to have installed a FLAS runtime. The Ziniki runtime is installed on top of (i.e. in the same directory as) the FLAS runtime, adding additional libraries, files and scripts.

First, download the latest `ziniki.zip` file from `the Ziniki Download Page`.

Unzip this into the same directory where you placed your FLAS runtime.

Check it was successful by identifying the existence of the `bin/ziniki` script:

```
$ ls -l bin/ziniki
…
```

When all this is done, you are in a position to compile the example (including running its unit and system tests) and run the example in a local container and access it from a browser.

If you haven't already (when installing FLAS) you want to add this `bin` directory to your `PATH` so that it can be run from anywhere.

1.5 Compiling Locally

Much of the time while developing, you will just want to compile locally and run the unit and system tests. Because you are using Ziniki functionality (such as the data service), you do need a full Ziniki instance installed locally, but you do not need to have the Ziniki server running (the compiler will run up mini Ziniki containers internally as needed).

To compile the code locally and run all the tests, do the following:

```
$ flas --web ui org.zinapps.samples.ziniki.hello
```

1.6 Running in a Browser

In order to be able to run Ziniki code in a browser, it is necessary to start a local microservice container. This is done using the `bin/ziniki` script.

It is not possible to run Ziniki directly as `localhost`. You must associate Ziniki with a domain for which you can generate or obtain a TLS certificate in a keystore. Doing this is outside the scope of this manual.First you must choose a domain that you want to use. For the purposes of this example, we will use `domain.com`.

You must make sure that requests on your machine remain local for the `ziniki` subdomain of your chosen domain, so add a line like this to your `/etc/hosts` (or equivalent) file:

```
127.0.0.1 ziniki.domain.com
```


As a complex piece of software, Ziniki has many options and can be configured in many ways; in particular, it has to be able to cope with different Cloud and DataCenter configurations, along with working with multiple different underlying storage infrastructures. All of this information is contained in configuration files, many of which are pre-canned and included with the distribution in the `config/` directory. The Ziniki script, by default, chooses the appropriate set to run a locally configured, all-in-memory instance. But there is still the final configuration which provides information about how the host should be configured.

This involves you creating a file somewhere on the file system similar to the domain.com.json file and providing appropriate values for the options.This example works with the optional `bootstrap.json` configuration file to make for simpler configuration.

```
$ ziniki --configFile config/local/bootstrap.json
--configFile config/dist/domain.com.json --user
https://ziniki.domain.com:18082/id/domainOwner --import-flim
flim
```

By default, tracing is largely suppressed (except for warning and errors) when using this script. It is possible to specify `--info` or `--debug` to obtain more detailed tracing information, and it is possible to specify a properties file for configuring tracing using `--trace=...`. For more information, see the reference manual.

The `--user` argument helps with the initialization process by creating the initial domain and the initial user without having to go through the complicated process of setting up an initial domain. It automatically creates the domain associated with the hostname for the user, and then creates a user credential and identity inside that domain. The user identity must be a URI; in the general case it can be any URI which identifies a user, but because we use the hostname portion to identify the domain, it must be in the domain you want to use.

Ziniki comes with a builtin OpenID authentication mechanism. With its default configuration, the ID as given here can be used for authentication, although this mechanism **does not** actually create such a user account in the authentication mechanism. However, if you create a new user with this id (specifically, the username "owner") through the authentication process, it will automatically have access to the Ziniki instance.

Because Ziniki runs in the foreground, you will need to open another terminal window in order to continue with this example.

In the new window, you need to "log in" using the zinlogin script[1].

```
$ zinlogin https://ziniki.domain.com:18083/
https://ziniki.domain.com:18082/id/domainOwner
```

The first argument here is the URI of the Ziniki admin server. As configured, that is the same host as the main server but with a port of 18083. The second argument is the identity URI as created on the `ziniki` command line above.

The result of this is to create a `.zintok` file in your home directory which can then be read by the `flas` compiler when it attempts to connect to Ziniki[2].

Then you need to compile the sample and load it into Ziniki. Do this by passing the appropriate value as the `--ziniki` argument to the `flas` command line as you compile the application:

```
$ flas --ziniki https://ziniki.domain.com:18080
--web ui org.zinapps.samples.ziniki.hello
```

When this has completed, the compiled code will have been stored within the local Ziniki instance, from which it can be accessed. In your browser, you can now visit

```
https://ziniki.domain.com:18080/app/org.zinapps.samples.ziniki.hello
```

[1] The zinlogin script does not go through the authentication process. It simply tells the Ziniki admin server that you want a login token and then stores it in the `.zintok` file in your home directory. While this may appear to be a security hole, this mechanism only works with the Ziniki admin server, which is not deployed in production instances.

[2] It should go without saying that there are other ways to create this file in production environments. This mechanism is provided for ease of local development, debugging and automation.

In the real world, you would probably configure your web server to deliver this application through a redirect from some path served by `www.domain.com`.

Because this loads the full experience, rather than the abbreviated test experience, you will need to log in before you can continue. Ziniki includes support for all OAuth and OpenID providers, but many of these require that a pre-established relationship has been configured before they will cooperate. Your memory-only Ziniki instance will not have any of these set up[3]. However, by default, when run from the command line Ziniki is configured to provide its own password-based OpenID Provider. This enables you to have a full login experience managed locally.

Since you already have a user, the easiest way to proceed is to create a login for that user. Click the "Log In" button presented on the home screen. Now paste the user credential from above into the "Other OpenID" box and click "Log In". This will redirect you through to the authorization server (you should notice the port has changed). Because you specified a complete credential, it is able to extract the username and requests you to enter a password and then press "New User".

The Ziniki authorization server does not have precise rules about what constitutes a password but demands a certain level of complexity. The more different character sets (letters, cases, numbers, symbols) that you use, the shorter the password can be; the longer a password is, the less diverse it needs to be. The maximum password length is 60 characters. For example, the password "domainOwner-has-a-password" will meet the minimum complexity requirements.

Because you are registering a user who is already in the Ziniki system (created in the startup of the in-memory Ziniki server above), there is no further registration required. Normally, it is necessary to configure the default user profile.

And finally you should be greeted.

Refresh the page and you should be told that you have been greeted two times.

[3] This is not to say that you couldn't configure it if you wanted to, but doing so is outside the scope of Ziniki documentation.

1.7 Full Stack Automated Testing

Once you have configured a standalone Ziniki server on your local machine, it is much as if you have truly deployed the application. As we have seen, it is possible to interact with the application in the browser.

Likewise, it is possible to test this application using technologies such as Selenium and WebDriverIO. In each case, it is necessary to configure your tests to start the Ziniki instance, load in the appropriate applications and then run your tests "in the normal way". These technologies are outside the scope of this manual but are supported and used internally as part of our continuous build process.

In later chapters, we will cover some of the additional techniques and functionality of the memory-only Ziniki server that enable you to do such testing more repeatably.

Commentary

1.1c The Ziniki DataStore

If all your previous experience of data stores revolves around relational databases and SQL, the way in which the Ziniki DataStore behaves will probably seem unusual and counterintuitive. In this case, you should probably switch your mental model and think about how programming languages frequently work.

There are a number of "well known", "named" or "identified" objects which can be recovered from the service "by name". We call these *roots*. Each root is like the root of a tree in a programming language: it acts as a starting point from which you can navigate the entire structure.

In this chapter, we have looked at the simplest form of root: for each user, it is possible to store exactly one object of a specific type in the store designated as `my` version of that object. Ziniki applications operate by having such an object which acts in the same way as a "home directory dot file" in a desktop application. The application defines a new type which contains all the core configuration information for the application (in the example, the number of times the application has previously said "hello"). On startup, the application recovers this object. If the user has never used this application before, then the object will not be there and the `missing` method is called, enabling the application to go into its default configuration flow and store the resulting object in the data store. On the other hand, if the object is there, the application can read in the previous configuration and follow any references to other objects, wherever they may be.

1.2c Entitites

The Ziniki data model is quite complex, particularly around its security model and constructs. But the basic element of most data modelling is the `entity`. In the simplest terms, an entity is just a `struct` with an identity.

A `struct` is just a value, so two structs are equal if and only if they have the same type and all of the members have the same value. But for dynamic systems, we need to be able to have entities whose values change over time. In order to do this, we need to have a different notion of equality: that the entities have an identity which is constant, regardless of how the value changes.

An `entity` in FLAS is defined to have two additional hidden fields which are automatically managed and populated by the system: an `id` and a `version`. The `id` is uniquely assigned by Ziniki when it first stores the object and the `version` is a monotonically increasing value which ensures that each version of the entity includes all the history of the object.

Because the `id` is only assigned when Ziniki first stores the entity, it is important to realize that an entity on construction in the client behaves much like a `struct` and it is important to ensure that you pass it around carefully and do not duplicate it. Once it has been stored and its identity established, it will only exist once in the client and will be shared rather than copied.

In this example, and it should be considered a pattern, we create and dispatch the new entity to Ziniki as quickly as possible, and then wait for the `value` to come back from the creation through the `value` method before using it to update the display. This avoids issues with multiple copies of the same entity.

The `id` of an entity in Ziniki is a URI. In fact, all `id`s in Ziniki are URIs. All entities have a scheme of `data` and their paths begin with `/entity`. The host is the domain of the server that created them. The rest of the path is sufficient to make the entity ID unique (but may not be unique among all Ziniki `id`s).

1.3c Overview of DataStore Structure

The exact format of the data store depends on the underlying infrastructure. But the abstract format is of a document database which resembles a map from URIs to JSON documents. This, in turn, is wrapped for the benefit of users by the `DataStore` contract which uses the concepts of *root*s, *link*s and *collection*s to build a representation of a data store akin to a `quiver`.

In this chapter we have used one root (the `my` record) and one link (from the `my` record to the entity). The `my` record is treated specially by the `DataStore` - it is a root - but it is nothing special internally: it is an entity resembling a JSON structure. Along with its `id`, it has a single field called `pointsTo` which is the `id` of the entity to be passed to the user. In processing a `my` request, the `DataStore` automatically dereferences this link and subscribes the user to the referenced object. Internally, the `DataStore` also subscribes to the `my` record and, if it should change, changes the subscription from the record originally pointed to, to the record now pointed to. The client is updated with the new value of the new object, but the fact that the pointer has changed is not mentioned.

Clients can only communicate over the `DataStore` protocol. However, System Tests can look "under the covers" at the actual contents of the data store. The `ziniki bind` command provides this functionality for entities - including `my` records - as we saw in the example test.

There is also a `ziniki dump` command which can be used within System Tests to show all of the entries in the data store. Note that because this shows *all* the entries in the data store, there will generally be many more than you are expecting, as all the entries which support the internal security model are present.

The Ziniki `DataStore` also enforces security, restricting who can access what elements. This model will be discussed more fully later, but the `my` record itself is exclusively available to the current user; or, put another way, each user has access to their own set of `my` records. This does not stop the entity pointed to being shared, but in general they will not be. Rather, the `my` record will point to a private "top-level" object which contains links to other objects which may be shared.

1.4c Subscriptions

It is a fact of life that things change. As data changes, applications should respond to those changes and update their display - automatically.

In normal programming models, this often seems insanely hard and ends up with multiple flows - one for when you want to fetch the data, and another for when it changes on you unexpectedly. Ziniki reverses this model and says that **whenever** you ask for something, you always want the most up to date value, you want to be kept up to date and the most up to date value will always be sent to the same handler. The FLAS language and programming model are designed to support this directly.

In the example, this is used to avoid duplication between the code in `missing` and `value`. The `missing` logic handles the case where the object does not exist. Its responsibility is to *make it exist*. It then stops. The value is created and populated on the server, which then turns around and delivers it to the (already existing) request for the most recent updates. From here the `value` logic takes over.

This same rule addresses a number of race conditions. Imagine, if you will, two clients both starting at the same time. They may both receive `missing` events and, in response, attempt to create a new object. Only one of them can succeed - it is not possible to create two `my` records. But, regardless of which of them succeeds, both will receive the `value` callback.

On the other hand, in this particular case, because we are trying to count the number of times the application starts, we have to be careful not to create an infinite loop by updating the counter **every time** we receive a `value` callback - because every time we update the value we will receive a new, updated value.

Subscribing to Non-Existent Roots

There is no rule that says you can only subscribe to things that already exist. By definition, every root has some kind of "name" - some way of accessing it that is independent of its existence. So when we subscribe to the `my` record for our counter, we do not have any expectation that it exists: indeed, that is why the `missing` method exists. But there is no need to implement the `missing` method. It would, for example, be perfectly reasonable to have two applications, one of which is responsible for generating data and another which is responsible for formatting it; the former would create the object but the latter would only consume it. Notwithstanding that, the consumer would be able to subscribe to the *name* of the root on startup; when the generator started and created the object, the consumer would automatically be notified and could react accordingly.

1.5c Ziniki and Apps

When compiling programs in FLAS without Ziniki, the result is a directory containing an HTML file and some javascript files. In addition to the runtime library, each package generates its own javascript files for the client code, Unit Tests and System Tests. The HTML file includes all of these in its headers and then creates the main card.

Ziniki operates in much the same way, except that none of the files are located on the file system. The HTML is generated on the fly in response to the query and contains links to all the assets and JavaScript files which are located in the Ziniki Content Store.

When compiling connected to Ziniki, the compiler first compiles all the code locally and runs all the tests to ensure that the code is valid. It then uploads all of the source code into the Content Store and creates an entity in the Data Store which represents the structure of the project and contains links to all the Content Objects. The compiler then asks Ziniki to build and deploy the code.

Ziniki validates the package structure and again compiles the code, reading it from the Content Store. Assuming that all the tests pass, it then stores all the generated assets in the Content Store and updates the project entity with the appropriate information before making it available to the system. It can then be loaded from browser clients.

Each package is versioned and depends on specific versions of other packages. When loaded and compiled, the code will always be compiled and linked against the *most current deployed* versions of the referenced packages. If these are not the versions you use locally, this can cause the local compilation to succeed but the Ziniki compilation to fail. Once deployed, the code will continue to depend on these exact versions, thus avoiding any problems with updating software.

The versioning is based on date, so multiple compilations on the same date (GMT) will cause only the last version to be persisted. In practice, this is unlikely to be relevant on local, memory-only instances of Ziniki; and most public instances are configured to not accept multiple same-day updates.

1.6c Domains and Users

Ziniki depends on the notion of *domains* in order to segregate content and servers. We haven't actually used the domain concept yet (we will talk a lot more about this in the next chapter), but one way in which domains segregate servers is that users are allocated on a per-domain basis, that is, each user is associated with a specific domain.

These domains are domains in the very familiar DNS sense. In short, Ziniki is assuming that in creating a Ziniki domain you are doing so in order to support an application which will be provided through or associated with that domain. The Ziniki server itself then assumes that its requests will be sent to `ziniki.<domain>` in the same way that web servers expect to be called `www.<domain>`. When processing requests, Ziniki will look at the `HOST` specified in the URI, check that it begins with `ziniki.` and then use the remainder of the host name as the domain associated with the request.

In the wild (i.e. when you start making your services public), Ziniki has to be sure that when you configure a Ziniki server you have the rights to do so by checking that you **also** control the web server for this domain. Again, we will look into this in more detail in a later chapter. The `--domain` option on the `ziniki` script bypasses this step, so it is not necessary to configure the actual web server for the requested domain.

For now, the most important thing is that you are able to create the appropriate entries in the DNS and certify them with a TLS certificate. The simplest way to do this locally is to edit `/etc/hosts` as we indicated above and provide a keystore that has at least some certificate for the `ziniki.<domain>` name you wish to use. Ziniki does not check the validity of the certificate, so self-signed certificates are fine providing that you have configured everything else (i.e. your browser) to tolerate them.

1.7c The `ziniki` Script and the Cloud

As noted above, the `ziniki` script can be configured to work in a number of different ways. The definitive information on how to configure the `ziniki` script is in the `Ziniki Reference Guide`. Although you should look there if you want full information on how to configure Ziniki, it seems relevant to say a few things here about configuring Ziniki for the Cloud.

In addition to configuring Ziniki to work in memory-only mode, it is possible to use the `ziniki` script to start a Ziniki instance running locally but with cloud-based resources.

The option `--aws` configures Ziniki to work with AWS Dynamo and AWS S3 as data and content storage mechanisms respectively.

By default, the `ziniki` script starts five servers concurrently: a `Ziniki` server, an `ID` and `Auth` server to provide OpenID logins, a content store and an admin service. It is possible to limit these by specifying one or more of the following options:

- `--id` starts the `ID` server;
- `--auth` starts the authentication server;
- `--ziniki` starts the main Ziniki server;

If any of these options are provided, then all the services mentioned explicitly will be started; while any services not mentioned will not be started.

It is not possible to start a fully cloud-aware Ziniki instance using the `ziniki` script; this is a more advanced process which may need manual configuration depending on your cloud environment. However, the Ziniki `$standup$` script enables you to relatively easily configure and deploy Ziniki within an AWS environment using `CloudFormation`.

While not all cloud environments are supported directly by Ziniki, the pluggable architecture and configuration model means that it is possible to extend Ziniki by implementing the various interfaces within Ziniki to support more environments and then to appropriately configure the resulting Ziniki instance.

Custom Ziniki components are outside the scope of this manual.

2 A Simple Shared Rating

In our most trivial example in the previous chapter, we did at least manage to build something that could be deployed to - and use - the cloud. The information that we kept about the number of times the user had interacted with Ziniki was stored persistently in the cloud, but was separate for each individual. What we want to do next is to allow multiple users to manipulate a shared object.

We are going to approach this through the metaphor of a simple thumbs-up/thumbs-down rating card. Each user is shown the same card backed by the same data object and has the opportunity to vote "up" or "down" on the service represented by the card[1]. The card maintains the overall rating of the service and displays that to each user.

So, without further ado, let's get to it.

2.1 The Visuals

First off, let's look at the visual component of this card. As you would expect, it is really quite simple.

```
<html>
<head>
</head>
<body>
    <div id='flas-card-ratings'>
        Current Rating: <div id='flas-content-current'
                              ></div>
        <div id='flas-style-up'/div>
        <div id='flas-style-down'/div>
    </div>
</body>
</html>
```

[1] As with all our demos, there is a lot missing here from a reasonable solution, such as the ability to restrict each user to one vote, keep the rating within bounds (eg. above zero) and express the rating in a user-friendly way (such as out of five). We will return to some of these issues later.

The card is placed in the main `div` with the `id` of `flas-card-ratings`. This has three elements: a `content` field which will display the current rating and two `style` fields which are the *up* and *down* buttons (we have used UTF-8 emoji for these). As usual, we have put no effort into styling these but the usual pattern applies: you can create and include CSS to your heart's content, and it will be deployed along with the card.

2.2 The Card

As with the previous example, we need an entity to store in the database to keep track of the current rating value. Even though this will be shared between many users, the code is just the same as before.

```
entity Rating
    Number value
```

The idea here is to keep track of the current rating as a single number: each up or down vote will change this number.

```
card Ratings
    state
        Rating rating

    template ratings
        current <- (show rating.value)
        up
            => thumbsUp
        down
            => thumbsDown

    requires org.ziniki.DataStore ds
```

The basic boilerplate of the card defines a `state` which will contain a `Rating` entity, a template which binds the current entity value to the `current` field in the HTML, and attaches an event handler to each of the emoji, and obtains a handle to the `DataStore` service in the variable `ds`.

```
    implements Lifecycle
        ready
            <- ds.domain domainName -> OnDomain
```



```
    domainName = "ratings.com"
```

When the card is ready to go, it requests a domain given in the variable `domainName`. It expects to find the ratings object attached directly to this domain.

```
        handler org.ziniki.DataHandler OnDomain
            missing
                <- Debug "There is a mismatched domain
                              name somewhere"
            value d
                <- ds.secondaryByType d (type org.ziniki.Dom
                              (type Rating) -> StoreRatingEn
```

If the code has been correctly deployed and set up, then the domain should be present. If the `missing` method is ever called, something has been misconfigured. In many cases, an entity being `missing` is merely a hint to create it, but it is not possible to create a domain from within the code, so we do the next best thing and respond with a message. By default, if `missing` is not implemented, the message is simply ignored, which is fine in this case, but slightly less clear. In development, this message will generally be visible (although it may be lost in other tracing); when deployed in the cloud, the message will only be visible to administrators.

When we obtain the domain, we attempt to recover the ratings object by using a *secondary key*. This is a way of naming items within a particular domain or arena, and is described in more detail in the commentary below. This will return the unique object to be given that name in this domain.

```
        handler org.ziniki.DataHandler StoreRatingEntity
            missing
                <- Debug "See Chapter 3"
            value r
                rating <- cast Rating r
```

The `missing` method in this case is entirely valid, but we don't want to tackle the complexities of creating and publishing this item just yet, so we are going to defer that until the next chapter. Instead, we will pre-populate the database with an appropriate object, pointed to by the appropriate secondary key. Just to make sure everything goes smoothly, if the `missing` method is called, we log an appropriate error.

When we receive back the value that we want, we store it in the appropriate state variable. As we do this, the display will automatically update with the current value.

```
event thumbsUp (ClickEvent ev)
    rating.value <- rating.value + 1
    <- ds.put rating

event thumbsDown (ClickEvent ev)
    rating.value <- rating.value - 1
    <- ds.put rating
```

The two event handlers are both basically the same, adjusting the (in state) value of the rating and then storing the updated version of the object back to the data store.

And that's it for the card, given the simplified set of requirements we are working with.

2.3 The System Test

In our opinion, this code is not complicated enough to need any Unit Tests. It's not clear what they would test beyond "if you click the button, it adds 1" which is what it says it does.

But we certainly want to be sure that the code does what we expect, so a system test is in order. We are going to take advantage of the storyboard nature of system tests to check that as we manipulate the card, the data is updated and the card with it. We will also check that multiple users can collaborate.

```
configure
    ziniki domainName
        datastore "datafiles"
        user "gareth"
    data Ratings card
    contract card Lifecycle ready
    data Ratings nelsonCard // but not initialized
```

In setting up this test, we configure the Ziniki embedded test server to operate on the domain defined in the test case, thus ensuring that we can change the domain by making the minimal number of changes.

This server is configured to use the datastore and to read files from the datafiles directory as discussed in the next section. We are ultimately going to use two users in this test, but for now we are going to start with a single user "`gareth`".

To finish the configuration, we create an instance of the `Ratings` card and call its `Lifecycle.ready` method.

```
test the initial rating is zero
    assert (card.rating.value)
        0
    match card text current
        0
```

Assuming everything has worked successfully, we should now have recovered the `Rating` entity from the datastore which, given the sample data we have set up, should have an initial rating of zero. Here we test both that the rating is zero and that this rating is showing in the content area.

```
test thumbs up increases the rating
    event card up (ClickEvent)
    assert (card.rating.value)
        1
```

The main thing we want to test is the functioning of the thumbs up and thumbs down buttons. Here we test that after simulating a press on the thumbs up button, the rating value stored on the card is 1.

```
test we can currently vote more than once
    event card up (ClickEvent)
    assert (card.rating.value)
        2
```

Although it is more a bug than a feature, we are currently able to vote more than once, so we make a test out of it. If we ever go back and fix this bug, this will give us an easy way in to drive the new behaviour.

Again, we simulate pressing the thumbs up button and confirm that the value is now 2.

```
test thumbs down drops it back to one
    event card down (ClickEvent)
```

```
        assert (card.rating.value)
            1
```

Here we simulate the pressing of the thumbs down button and then assert that the resulting rating is 1.

```
    test a second user initially sees the same value
        ziniki domainName
            user "nelson"

Only now do we initialize the second card

        contract nelsonCard Lifecycle ready
        assert (nelsonCard.rating.value)
            1
        match nelsonCard text current
            1
```

A key part of this test - indeed, this whole chapter - is the notion that it is possible to have entities shared between multiple system users. So now we are going to create a second user - and have them create their own card - and observe the interactions between the two.

The first step is to switch the user. The `ziniki` command ensures that the specified user - `nelson` exists in the system and asserts that any new cards will connect as this user. The next two steps create a card for this user and initialize it in the same way as in the configuration step.

Finally, we assert that the card has initialized correctly with the same rating, both in the entity and on the screen.

The newly created user now simulates pushing the thumbs up button. This increases the rating to 2, which we test.

```
    test the first user is notified of the change
        assert (card.rating.value)
            2
        match card text current
            2
```


Finally, since this object is shared through the Ziniki server, when the second user saves their rating (using the `ds.put` method), the first user should be notified of the change and receive the updated value and display it without needing to do anything.

2.4 Configuring the System Test

In order to be able to defer the creation of the shared entity until the next chapter, we need to configure this test to have that object where we need it before the test starts.

When we configure the Ziniki server, we declare the data store to read the contents of a directory, in this case `datafiles`. As we saw in the previous chapter, if this directory does not exist, then it is simply ignored (although the data store is created and made available).

The contents of this directory are the hierarchical contents of the server, separated by domain name.Within each of these directories are a set of files. Each file corresponds with one document in the data store. The file extension identifies what *kind* of object it is and the name is used as part of the object identifier.

In this case we have two objects we want to create: one is the entity itself (entities have the extension `.en`) and the other is a secondary key record pointing to the entity.

```
    type org.zinapps.samples.ziniki.ratings.Rating
        number "value" 0
```

This file follows a FLAS-like format - using significant indentation - to define and configure the object.

For an entity, there is only one top level command, which is the `type` command. This specifies the (always fully qualified) name of the type of the object.

Within this are the values of the object fields. Each field must be of an appropriate type and specify a field name as a string and a value.

In this case, there is only one field - the current rating `value` - which is set to the `number 0`.

This object is stored in the key `data://ratings.com/entity/rating`. The scheme (`data`) and the first portion of the path (`entity`) come from the fact that it is an entity with the `.en` extension; the domain comes from the parent directory and the final element of the path is taken from the object name.

Secondary key catalogues always have the extension `.sk`, while their name is the name of the key itself. Inside the file they have a format that identifies each key in turn and its corresponding value.

This is a trivial secondary key because it does not have any additional fields - there is only one such key in each domain or arena. As indicated by the comments, if the key catalogue did have additional fields, they would be specified by the `field` top-level command.

Each entry in the catalogue has a pair of `key` and `value`. The `key` must have the same number of nested `field` elements as the catalogue has top level `field` elements. The value will always have one string entry which is the `id` of the entity referenced by the secondary key.

In this case, there are no `field` commands either at the top level or nested within the `key`. The value is defined to point to the entity we created earlier.

2.5 Compiling the Example

Compiling this example, and deploying to Ziniki, is just like the previous chapter. Refer to that for details on how to start the ziniki process and create a simulated login for the compiler to use.

Compile and deploy using the following command:

```
$ flas --ziniki https://ziniki.domain.com:18080
--web ui org.zinapps.samples.ziniki.ratings
```

This should compile smoothly, run through the system test, upload to the Ziniki server and deploy the application.

2.6 Running in a Browser

Unless you happen to own ratings.com - or you are happy working with self-signed certificates - you will need to use a different domain: one for which you do have a certificate (see the previous chapter for configuring that).

In order to change the domain, you need to change three things:

- the value of the `domainName` variable in ratings.fl
- the subdirectory of `datafiles` must have the appropriate domain name
- the `value` in the `sk` file must have the same domain name as the host name in the URI

As with the system test, until we add the ability to create `Rating` entities in the next chapter, we need to configure the data store with the same initial objects that we used for the system test.

This is easily done by passing the `--datafiles` argument to the `ziniki` script. Note that this must be done *after* the initial users (and their corresponding domains) have been configured using the `--user` arguments; the loading process uses the set of currently configured domains to determine which subdirectories of the `datafiles` directories to read, and if a domain is not configured, the corresponding directories will be ignored.

```
$ ziniki --configFile config/local/bootstrap.json
--configFile config/dist/domain.com.json --user
https://ziniki.domain.com:18082/id/domainOwner --import-flim
flim --datafiles datafiles
```

This assumes that the `datafiles` directory is in the directory in which the script is invoked, but obviously a more complete path could be used.

It should now be possible to visit the application by deploying it as described above, and visiting:

```
https://ziniki.domain.com:18080/app/org.zinapps.samples.zini
```

As in the previous chapter, this requires you to complete the login process (creating a login for the new user `domainOwner` with a suitable password) and then use the thumbs up and thumbs down buttons.

To see the multi-user behaviour, it is necessary to access the card in a separate browser context (it is important that no user state is shared between the two; a different browser is the easiest approach). You will then need to log in as a completely new user, and follow the full registration process.

Once complete, the same rating card - with the most up-to-date values - should appear. Subsequently, as you interact with either card, the other should update in real time.

Commentary

2.1c Sharing between Users

In the previous chapter, we created a private object. In this chapter, we have used a shared object (and in the next chapter we will see how we create and share objects). Ziniki is (almost) equally happy working with private and shared objects.

All shared objects are stored within a tree-like structure rooted in a *domain*. Each domain must be configured through the administration UI. Within the domain, entities constitute the leaf nodes, while junction nodes are implemented by *arena*s (covered in some later chapter). Access to an entity or an arena is determined by the immediately containing domain or arena.

2.2c Finding Shared Objects

In the previous chapter, we discussed the `my` method and how that enables private entities to be found by type. In this chapter, we have found a shared entity using two techniques. First, we used the `domain` method to find a domain using its *name*, and then we used a *secondary key* to find the unique rating object in that domain.

A secondary key catalogue is configured with a unique name and a set of fields. Although the catalogue may appear to be defined once, when it is attached to an arena it is independent from all the other instances attached to other arenas.

Each secondary key must have a unique value for a specific combination of values for the key fields within the same arena or domain. In our case, there are no field values, so this limits the number of possible values to just the one - the one we want. But we could configure this to have one or more fields - such as the name of a restaurant to rate, as we will see in a later chapter - and use that to maintain and find multiple `Rating` entities in the same domain. But it is worth pointing out that it would be possible to configure the same secondary key catalogue in a different domain to support multiple objects (one for each domain).

3 Creating a Shared Object

In the previous chapter, we worked with shared objects, but deferred the actual creation of such an object. In this chapter, we are going to enhance the previous example by handling the case where the requested object turns out to be missing.

3.1 The Code

The visuals, the system test and most of the code is exactly the same as in the previous chapter.

The first change is in `OnDomain`. In the previous chapter, we used the domain directly to create the secondary key.

```
handler org.ziniki.DataHandler OnDomain
    missing
        <- Debug "There is a mismatched domain
                    name somewhere"
    value d
        <- ds.secondaryByType d (type org.ziniki.Dom
                    (type Rating) -> StoreRatingE
```

Now, we need to use the domain twice: once to create the secondary key and once to pass to the handler which will publish the new entity rating to the domain. Consequently, we extract it into a variable that can be shared between the two. At the same time, we cast it to a `Domain` to improve type checking.

```
handler org.ziniki.DataHandler OnDomain
    missing
        <- Debug "There is a mismatched domain
                    name somewhere"
    value v
        <- ds.secondaryByType d (type org.ziniki.Dom
                    (type Rating) -> (StoreRatingE
                    d)
            d = cast org.ziniki.Domain v
```

The most significant change is in the `StoreRatingEntity`. In the previous chapter, we had merely logged an error when the `missing` method was called. Now we want to create a new object and attach it to our "personal" arena. We also provide a handler which will receive the object once it has been created.

```
handler org.ziniki.DataHandler StoreRatingEntity
                    (org.ziniki.Domain d)
    missing
        <- ds.attachPersonalArena (Rating 0) ->
                    ShareRating d
    value r
        rating <- cast Rating r
```

The `ShareRating` handler is new.

```
handler org.ziniki.EntityCreated ShareRating (org.ziniki.Domain
                    d)
    created e
        <- ds.publish (cast Rating e) d ["public"]
```

This handler implements a different contract to the other handlers we have seen so far, but the `created` method is comparable to the `value` method in the `DataHandler` contract. It is, however, not possible for the `attachPersonalArena` to return a `missing` method since it is creating a new object not searching for an existing one.

The variable `r` is introduced to provide a type-safe version of the entity received on creation. This should be essentially the same object that we passed to `attachPersonalArena` but it will have been assigned an `id` in the same way that the `putMy` method assigned an `id` to the object created in Chapter 1. This is then immediately assigned to the `rating` object.

The final step is to *publish* this entity. Publication[1]is the process by which an entity to which you already have access is made available in a different context. The simplest such operation is to take an object which is attached to your personal arena and attach it to a shared arena. Domains are special cases of arenas.

[1] There is a lot more information about how this works in the commentary and will be amplified in subsequent chapters.

The act of publishing an entity involves specifying the entity, the arena to which it is to be published and the benches (in that arena) to which it is attached. Here we specify the entity we received back, the domain we recovered previously and the `public` bench.

Everything else remains the same.

3.2 Removing the Configuration

Because we are now creating the shared entity, it no longer needs to be created by default in the `datafiles` configuration. Consequently, we have deleted these configuration files.

Commentary

3.1c Arenas and Domains

In the previous chapter, we briefly started to touch on the subject of shared objects and the primary sharing mechanism in Ziniki, the *arena*.

An *arena* is not dissimilar to a folder or directory in a classic operating system: it is a way of organizing content and controlling access to it. Like directories, arenas may be nested. In the same way, a *domain* corresponds to the root of such a directory hierarchy, containing other arenas and entities.

Access to an arena is controlled through the concept of *bench*es. Each entity in the arena may be attached to one or more of the *bench*es defined in the arena. Each bench has a specific set of permissions - actions that can be carried out either on the attached entities or on the arena itself and its benches - and a set of *persona*s (each associated with a single identity) who are granted those permissions.

In this way, the arena, and in particular the benches of an arena, constitute the central access control mechanism in Ziniki. The other one that we have met so far is the "personal arena" which is the same in principle, but only provides access to your own objects.

3.2c Creating Entities

Creating entities is just like creating structs. They are created within actors using constructor syntax. But then they must be saved to the data store in some way. One approach (as followed in Chapter 1) is to save an individual entity as a rooted `my` object, which can be recovered by type using the `my` method. A second approach, introduced here, is to pass the newly created entity to the `attachPersonalArena` method, which ensures that the entity is assigned a unique ID and attached to the current user's personal arena, from which it can be recovered by iteration.

3.3c The Personal Arena

From what we have seen so far, it might seem that the "personal" arena is very different from the standard, shared arenas. However, that is largely because everything we have seen so far have been the points of difference. Otherwise, a personal arena has almost exactly the same capabilities as a shared arena.

To clarify, the differences are:

- The personal arena has exactly one bench and no more can be added;
- That bench has the owning user on it and no other users may be added;
- The bench has unlimited permissions on all the objects within it;
- The personal arena may not have any nested arenas.

Technically, objects that are stored with the `my/putMy` mechanism are not stored in the personal arena but just in a personal space which cannot be accessed by other individuals. If desired, however, the same entity may be later attached to the personal arena and even published; this is not, however, a common or recommended usage pattern.

3.4c The Owner and Public Benches

Two benches are automatically created for every arena with pre-defined privileges: the `owner` bench has all the privileges, just like the bench in the personal arena and the user creating the arena is automatically added to it; the `public` bench has just the `ReadEntity` permissions and does not have - and cannot have - any users specifically added to it, but all users of the system *implicitly* have access to any entities or arenas attached to the `public` bench.

3.5c Publishing

As noted above, entities are created in "client space" and are initially not stored in the data store. However, once an object is created, it may be stored and attached to a user's personal arena. Once here, it may be *published* to any shared arena for which the user has the appropriate permissions.