Seed Tutorial

This tutorial will introduce you to Seed and its prototype. You will learn how to build services, use the service monitor, apply transformations, write a browser-based client, and write map and reduce functions. The working example for this tutorial is a key-value store (kvs).

1 Prerequisites

Several things are needed before we can get started.

- Go `http://golang.org`
- Gnuplot
- Seed `https://github.com/nathankerr/seed`
- A web browser (tested in Safari, Chrome, and Firefox)
- Some knowledge of command lines, go, javascript, html, websockets, json, and services.

Seed will work on Windows, OS X, and Linux. The command line instructions are geared toward OS X and Linux. If you are using Windows, please use `cmd` or write `%--` before command line arguments (if using PowerShell). Also, change `/` to `\`.

2 Getting Started

Start by making a new directory to work in:

```
mkdir seed-tutorial; cd seed-tutorial
```

Create a file, `kvs.seed`, with the following contents (don't include the line numbers).

```
1 table kvstate [key] => [value]
```

This file defines the core of a key-value store service. Line 1 describes a table called `kvstate` with a single key column called `key` and a single value column called `value`.

3 Starting the Service

Start the service by running:

```
seed --sleep 0.5s --monitor :8000 --execute kvs.seed
```

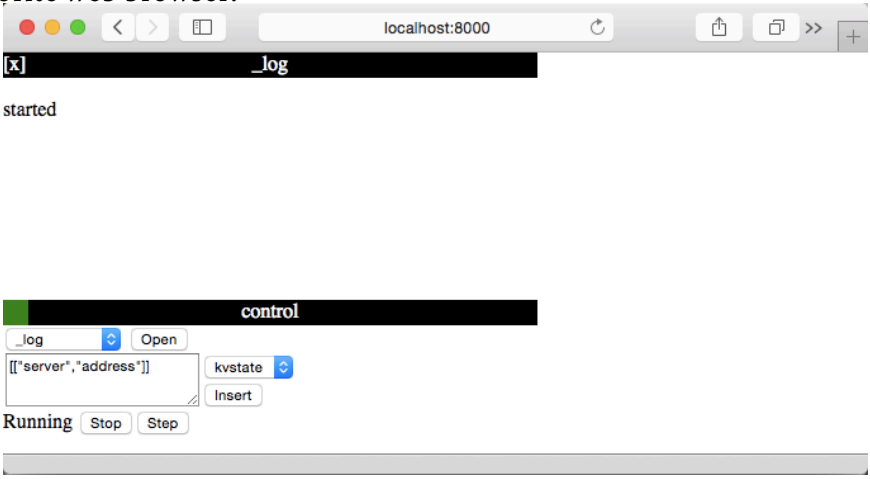
`--sleep 0.5s` will sleep half a second in between time steps. This slows things down enough so that we can see what is happening in the service instance. A sleep value is needed when using the monitor to limit the data the web browser must process.

`--monitor :8000` starts the debugging monitor on port 8000. To restrict access to localhost, use `--monitor localhost:8000`.

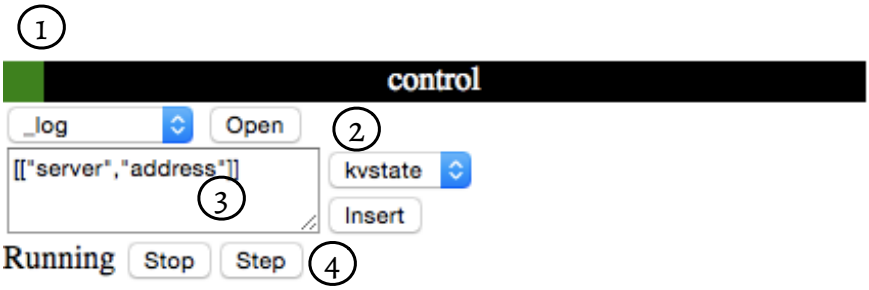
`--execute` starts the service using the built in interpreter.

4 The Monitor

The Seed prototype includes a monitor that allows you to see what is happening in your service. Now that the service has been started, the monitor can be accessed at `http://localhost:8000` using your favorite web browser:



The debug monitor presents a view of the state of the service instance. The control area has controls for controlling the monitor.



1. Green shows the monitor is connected; red disconnected.
2. Access to available views. Select the desired view and press open.
3. Insert data into a collection. Data is encoded in JSON as an array of arrays. "server" and "address" are special values that are translated into the server and client addresses respectfully.
4. Execution control. Start and stop the service, and step through the parts of the time step.

The rest of the monitor area is for the views. Click on the name of the view to put it in the main area (on the left, where _log is); [x] to close the view. Views beginning with underscores are special.

_log The monitor log

_service The service that is running (expressed in Seed)

_graph The flow graph of the running service. Rule numbers in the graph correspond to those in _service.

_time The time spent on the last timestep

Views not beginning with underscores correspond to collections (e.g., tables, inputs, outputs) in the service.

You can use the monitor to insert data into kvstate. The displayed order of rows for kvstate might fluctuate. This is due to the unordered nature of collections[1].

You can stop the service at any time by pressing ctrl-C in the command line.

5 KVS Put Operation

We will now expand the core of our kvs service by adding a put operation. Put inserts key-value pairs into kvstate.

Update kvs.seed to match:

```
1 table kvstate [key] => [value]
2
3 input kvput [key] => [value]
4 kvstate <+- [kvput.key, kvput.value]
```

Line 3 defines an input called kvput that contains two columns: key and value. Like all collections, inputs have key and data columns. Input and output collections define the flow of data into and out of a service. These collection types are not network accessible.

Line 4 is a rule which projects the columns in kvput and then upserts (<+-) the resulting set into kvstate.

Start the service with:

```
seed -sleep 0.5s -monitor :8000 -execute kvs.seed
```

Insert stuff into kvput and watch how it flows to kvstate. Try using various values to see how things work. Also look at the _service and _graph views to see how they relate. Note that no transformations have been applied (we will transform the service in later steps).

6 Put Client

We will now build a client for the service. First the service needs a network interface. When starting the service, the network interface

[1]Collections are implemented using Go's map datatype. When Go maps contain a small number of keys, the order is randomized when accessing all the keys so that developers will not accidentally depend on the implementation's natural ordering of this unordered dataset.

is added by using the network transformation:

```
seed -sleep 0.5s -monitor :8000 -execute
    -transformations network kvs.seed
```

Check the service's _service and _graph views to see the results of the transformation.

The client will use a custom json over webservices protocol. This protocol was implemented to allow web browser based clients.

The client is implemented as a static web page (index.html):

```
<!DOCTYPE html>
<html>
<head>
<title>Put Client</title>
<meta charset="utf-8" />
<script>
  var websocket, address, connected, server
  var key, value

  function onClose() {
    connected.style.backgroundColor = "red"
  }

  function send() {
    try {
      websocket.send(JSON.stringify({
        Operation: "<~",
        Collection: "kvput",
        Data: [[server, key.value, value.value]]
      }))
    } catch (e) {
      alert(e)
    }
  }

  function init() {
    connected = document.getElementById("connected")

    key = document.getElementById("key")
    value = document.getElementById("value")

    // connect to the monitor server
```

```
33      server = "ws://" + window.location.hostname +
            ":3000/wsjson"
34      websocket = new WebSocket(server);
35      websocket.onclose = onClose;
36
37      // set uniq id (address)
38      websocket.onopen = function() {
39        address = Math.random().toString(36).substr(2)
40        try {
41          websocket.send(JSON.stringify(address))
42        } catch (e) {
43          alert(e)
44        }
45
46        connected.style.backgroundColor = "green"
47      }
48    }
49
50    window.addEventListener("load", init, false)
51  </script>
52  </head>
53  <body>
54  <div id="connected" style="width:20px; height:20px;
        background-color:red"> </div>
55  <div>
56    <form onsubmit="return false;">
57      <input type="text" id="key" value="Key" />
58      <input type="text" id="value" value="Value" />
59      <input type="submit" value="Put"
            onclick="send()"/>
60    </form>
61  </div>
62  </body>
63  </html>
```

The client is initialized with the init function on Lines 26-48 this is ran on load (Line 50). A websocket connection is setup in Lines 32-35. A pseudorandom address is then created and sent to the server. This address will be used as the client address for Channel based communication and lasts as long as the websocket.

When initialization is completed, the connection indicator (Line 54) is turned green. When the websocket is closed, the connection will

turn red (Line 35, Lines 10-12).

When the put button in the form is pressed (or the form is otherwise submitted), the send function in Lines 14-24 is executed. This function simply builds, serializes, and sends the data over the websocket to the server.

Because web browsers don't like allowing websocket access from pages loaded from the local file system, we need a simple web server:

```
1  package main
2
3  import (
4    "fmt"
5    "net/http"
6  )
7
8  const addr = ":4000"
9
10 func main() {
11   http.Handle("/", http.FileServer(http.Dir(".")))
12
13   fmt.Println("Starting Put client on", addr)
14   err := http.ListenAndServe(addr, nil)
15   if err != nil {
16     println(err.Error())
17   }
18 }
```

Start the client web server by:

```
go run server.go
```

And then open your browser to `http://localhost:4000`. As usual, the server will bind to all your computer's network interfaces. To restrict it, change addr on Line 8 to localhost :4000.

To see a more advanced client that also handles results, look at the time service client's implementation.

7 Replication

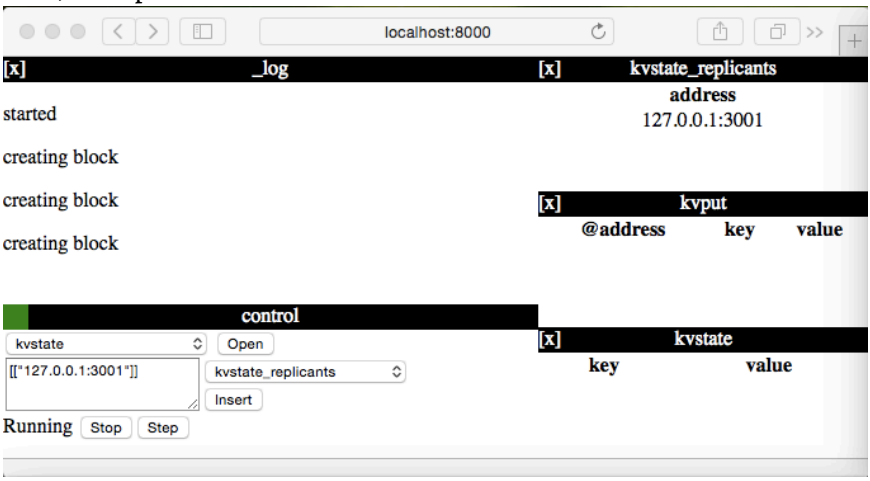
We will now setup and run a replicated kvs. No changes to any code are needed to do this.

Start the first replica:

```
seed -sleep 0.5s -monitor :8000 -execute
    -transformations "network replicate" -address
    :3000 kvs.seed
```

Insert [["127.0.0.1:3001"]] into :8000's kvstate_replicants using the monitor.

Also, set up the views as follows:

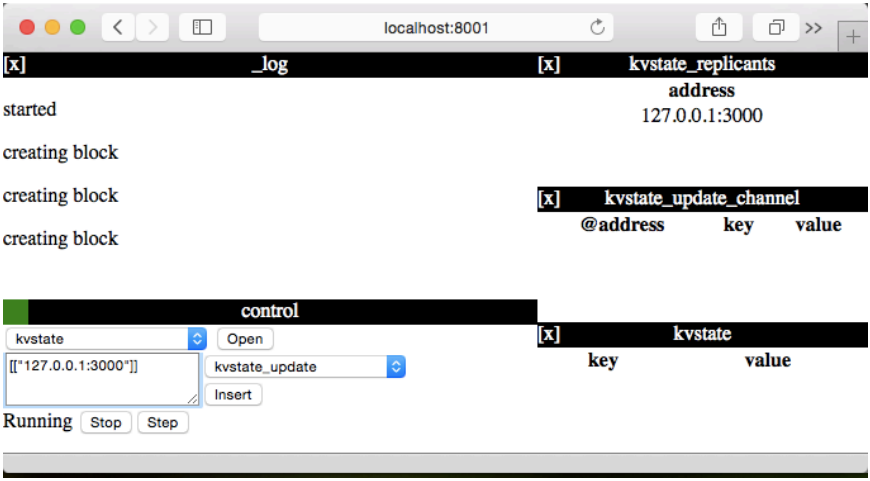


Start the second replica:

```
seed -sleep 0.5s -monitor :8001 -execute
    -transformations "network replicate" -address
    :3001 kvs.seed
```

Insert [["127.0.0.1:3000"]] into :8001's kvstate_replicants using the monitor.

Also, set up the views as follows:



Start the client (will connect to :3000):

```
go run server.go
```

If you have the two replica monitors and the client in different windows, you can watch the entire progression.

Put some value into the client and watch as it first hits the first replica and then the second.

Note that due to the transformations, no changes were needed to kvs.seed or the client to enable replication. However, while the client will only connect to one of the replicas, this could be resolved by using a load balancer.

8 Compiling Services

Up to this point the services have been interpreted by the seed tool. Now we will compile the service. This is done in two steps:

1. Use the seed tool to perform desired transformations and output go source code for the service.

2. Use go to compile the resulting code.

This process will be done for the replicated kvs. First,

```
seed -transformations "network replicate" -t go
    kvs.seed
```

applies the network and replicate transformations a before. However, all execution options have been removed and replaced with `-t go`, which tells `seed` to output `go` source code. By default this is done is a directory called `build`. To compile your replicated kvs:

```
cd build
go build -o kvs
```

This first moves to the build directory and then uses `go build` to build all the `go` files in that directory and name the resulting executable `kvs` (the default is the name of the directory).

The service can then be started with:

```
./kvs -sleep 0.5s -monitor :8000 -address :3000
```

The same execution options (e.g., sleep, monitor, address) can be used as before.

9 Map

We will now extend the service by using a map function. Map functions are not implemented in Seed, but in the host language (in this case, `go`). As the function is implemented in the host language, it can do any things the host language can.

The map function we will build converts a string to upper-case characters. First modify `kvs.seed` to:

```
1 table kvstate [key] => [value]
2
3 input kvput [key] => [value]
4 kvstate <+- [kvput.key, kvput.value]
5
6 table upper [key] => [value]
7 upper <+- [kvput.key, (upper kvput.value)]
```

Line 6 defines a new table where we will keep the key, uppercased-value pairs. Line 7 take values from `kvput` like we did on Line 4, but instead of using the raw `kvput.value`, it runs a map function `upper` with a single argument `kvput.value`. Map functions and their arguments are surrounded by `( )`. Arguments are space delimited.

Define the map function in `upper.go`:

```
1  package main
2
3  import (
4    "strings"
5
6    "github.com/nathankerr/seed"
7  )
8
9  func upper(input seed.Tuple) seed.Element {
10   return strings.ToUpper(input[0].(string))
11 }
```

Line 1 declares the contents of the file to be part of the main package. Lines 3-7 import the two packages we will use to define the map function. We will use `strings.ToUpper` to uppercase `kvput.value`. The seed package is used to access the types needed by a map function. A map function is defined as `func(Tuple) Element`. A `Tuple` is `[] interface {}` and will contain the arguments to the map function for that row. An `Element` is `interface {}` and will be used in the place of the map call.

The function itself is deceptively simple. `input[0]` grabs the first value in the input tuple. In this case, there should only be one value, as we only specified one argument to `upper`. If there were more arguments, they would appear in `input`.

`input[0].(string)` asserts that `input[0]` is a string. This is needed because a `Tuple` can contain any type. The actual type is dependent on the data sent to the server (JSON over websockets) and the type the JSON unmarshaller uses for that data. If some data that is not unmarshalled as a string, such a number, is there, the server will panic. Thus a better implementation would need to handle those cases.

Finally, `strings.ToUpper` converts that value to its upper case value, which is then returned. The executor uses the returned value to fill the map function's place in the result tuple before inserting it into `upper`.

The whole build and run process can be done as follows:

```
seed -t go -transformations network kvs.seed
cp upper.go build/
cd build
go build -o kvs
```

```
./kvs -sleep 0.5s -monitor :8000
```

Use the monitor (and the client if you like) to see that the map function is called and its results.

10 Reduce

We will now use a reduce function. Reduce functions are similar to map functions, except they take an array of tuples instead of a single tuple as input.

The example we will build counts the number of keys that have the same value.

First modify kvs.seed:

```
1 table kvstate [key] => [value]
2
3 input kvput [key] => [value]
4 kvstate <+- [kvput.key, kvput.value]
5
6 scratch count [value] => [count]
7 count <= [kvstate.value, {count kvstate.key}]
```

Lines 6-7 define our new collection and rule. A scratch collection is used so that other operations on kvstate such a removing tuples will not cause incorrect values in our count collection.

The syntax for the reduce function count is similar to that of a map function, except that { } are used instead of ( ). Fields in the intension other than the reduce function group[2] the rows passed to the function. In this case, all the kvstate.keys that have the same kvstate.value are passed to the count reduce function. count is called for each unique kvstate.value.

The reduce function is implemented in count.go:

```
1 package main
2
3 import "github.com/nathankerr/seed"
4
5 func count(input []seed.Tuple) seed.Element {
```

[2] Similar to how GROUP BY is used in combination with an aggregate function in SQL.

```
6 |   return len(input)
7 | }
```

A reduce function is a `func([]Tuple) Element`. Lines 5-7 implement `count`. In this example we don't have to worry about the data in the slice of tuples, as we just want its length (count). Line 6 returns the length of the input slice.

Building and running the service can be done as follows:

```
seed -t go -transformations network kvs.seed
cp count.go build/
cd build
go build -o kvs
./kvs -sleep 0.5s -monitor :8000 -address :3000
```

Use the monitor (and the client if you like) to explore the behavior of the reduce function.