# Persistent Storage with JSON

This lesson explains how to build persistent storage for the application using JSON.

WE'LL COVER THE FOLLOWING



## Using JSON for storage #

If you are a keen tester you will perhaps have noticed that when `goto` is started two times, the 2nd time it has the short URLs and works perfectly. However, from the 3rd start onwards, we get the error: `Error loading URLStore: extra data in the buffer`. This is because *gob* is a stream-based protocol that doesn't support restarting. We will remedy this situation, here, by using JSON as a storage protocol, which stores the data as plain text, so it can also be read by processes written in other languages than Go. This also shows how easy it is to change to a different persistent protocol because the code dealing with the storage is cleanly isolated in 2 methods, namely `load` and `saveLoop`.

Start by creating a new empty file **store.json**, and change the line in **main.go** where the variable for the file is declared:

```
var dataFile = flag.String("file", "store.json", "data store file name")
```

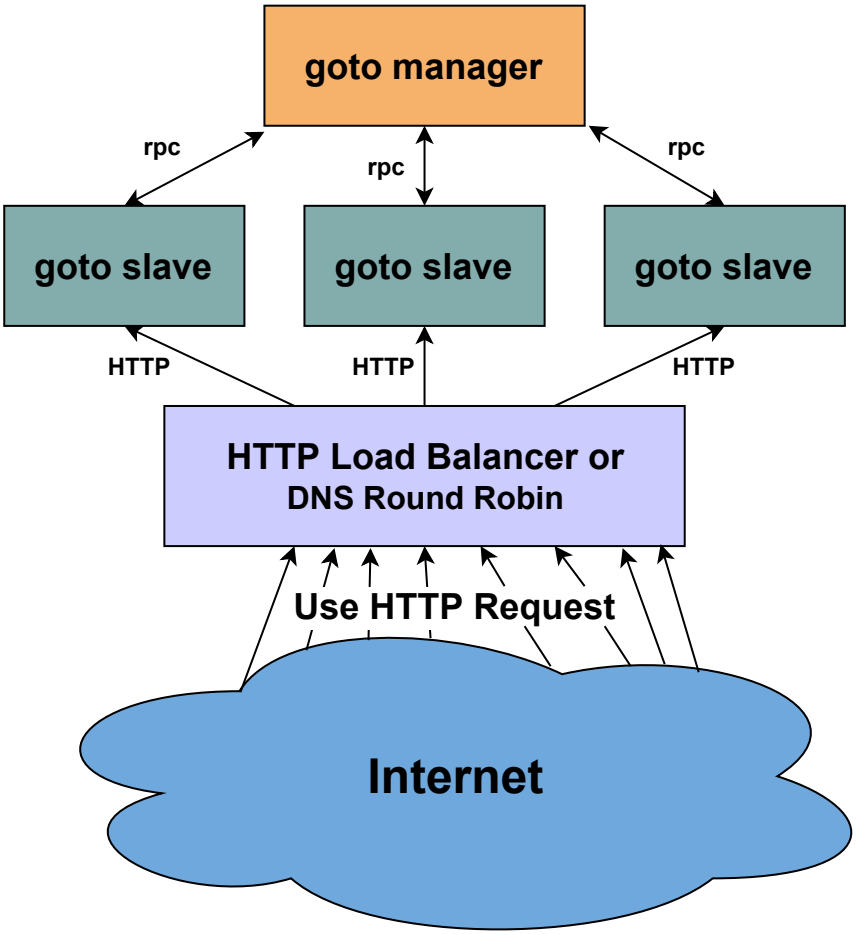
In **store.go**, import `json` instead of `gob`. Then, in `saveLoop`, the only line that has to be changed is: `e := gob.NewEncoder(f)`. We change it in: `e := json.NewEncoder(f)`. Similarly, in the `load` method, the line `d := gob.NewDecoder(f)` is changed to `d := json.NewDecoder(f)`. This is everything we need to change! Compile, start and test; you will see that the previous error

does not occur anymore.

As an exercise, write a *distributed version* of the program, as described in the following section.

## Multiprocessing on many machines #

Until now, `goto` runs as a single process, but even using *goroutines*, a single process that runs on one machine can only serve so many concurrent requests. A URL shortener typically serves many more `Redirects` (reads, using `Get()`) than it does `Adds` (writes, using Put()). So, it is better to create an arbitrary number of read-only slaves that serve and cache `Get` requests, and pass `Puts` through to the master, like in the following schema:




Distributing Workload over Master-and Slave Nodes

To run a *master* instance of the application `goto` on another computer in a network other than the *slave* process(es), they have to be able to communicate with each other. Go's `rpc` package provides a convenient means of making function calls over a network connection, making `URLStore` an `RPC` service. The slave process(es) will handle `Get` requests to deliver the long URLs. When a new long URL has to be transformed into a short one (using a `Put()`

method), they delegate that task to the master process through an RPC-connection; so, only the master will have to write to the data file.

The basic `Get()` and `Put()` methods of `URLStore`, until now, have the signatures:

```
func (s *URLStore) Get(key string) string
func (s *URLStore) Put(url string) string
```

RPC can only work through methods with the signature (t is a value of type `T`):

```
func (t T) Name(args *ArgType, reply *ReplyType) error
```

To make `URLStore` an RPC service, we need to alter the `Put` and `Get` methods so that they match this function signature. This is the result:

```
func (s *URLStore) Get(key, url *string) error
func (s *URLStore) Put(url, key *string) error
```

With this changed signature in mind, write the code for the new `Get()`, `Put()` and `Set()` functions. Don't forget to adapt the call to `Set()` from `load()` to `s.Set(&r.Key, &r.URL)`, because the arguments are now pointers. You also have to modify the HTTP `Redirect` and `Add` handlers to accommodate the changes to `URLStore`.

Add a command-line flag to enable the RPC server in main(), as follows:

```
var rpcEnabled = flag.Bool("rpc", false, "enable RPC server")
```

To make RPC work, we have to register the `URLStore` with the `rpc` package and set up the RPC-over-HTTP handler with `HandleHTTP`, like this:

```
func main() {
  flag.Parse()
  store = NewURLStore(*dataFile)
  if *rpcEnabled { // flag has been set
    rpc.RegisterName("Store", store)
    rpc.HandleHTTP()
```

```
  }

  ... (set up http like before)
}
```

## Using a `ProxyStore` #

- Now that you have the `URLStore` available as an `RPC service`, build another type (call it `ProxyStore`) that represents the rpc-client and that will forward requests to the RPC server:

  ```
  type ProxyStore struct {
   client *rpc.Client
  }
  ```

  The goal is that the slave(s) use the `ProxyStore`, only the master uses the `URLStore`.

- Make a function `NewProxyStore` that creates our `ProxyStore` object. An RPC-client is created in that function by using the `DialHTTP()` method to connect to an RPC-server. This `ProxyStore` also needs `Get` and `Put` methods that pass the requests directly to the RPC server using the `Call` method of the RPC client:

  ```
  func (s *ProxyStore) Get(key, url *string) error {
    return s.client.Call("Store.Get", key, url)
  }
  func (s *ProxyStore) Put(url, key *string) error {
    return s.client.Call("Store.Put", url, key)
  }
  ```

- In order for the slaves to handle the `Get` requests, they must have a copy (a cache) of the `URLStore`. So, add a pointer to `URLStore` to the `ProxyStore`. Adapt the function `NewProxyStore`.
- Then, modify the `NewURLStore` function so that it doesn't try to write to or read from disk if an empty filename is given.
- Expand the `Get` method, so that it first checks if the key is in the cache. If present, `Get` returns the cached result. If not, it should make the RPC call, and update its local cache with the result.
- Similarly, the `Put` method needs to only update the local cache when it performs a successful RPC-Put.

- You perhaps realized that `URLStore` and `ProxyStore` look very much alike: they both implement `Get` and `Put` methods with the same signature. So, try to specify an interface `Store` to generalize their behavior. Now, our global variable store can be of type `Store`:

```
var store Store
```

- Finally, adapt the `main()` function so that either a slave- or a master process is started up (and we can only do that because store is now of the interface type `Store`). To that end, add a new command line flag:

```
var masterAddr = flag.String("master", "", "RPC master address")
```

with no default value. Adapt `main()` so that if a master address is given, start a slave process and make a new `ProxyStore`; otherwise, start a master process and make a new `URLStore`. Now, you have enabled `ProxyStore` to use the web front-end, instead of `URLStore`. The rest of the front-end code continues to work as before. It does not need to be aware of the `Store` interface. Only the master process will write to the data file. Now, we can launch a master and several slaves and stress-test the slaves.

Compile this version like the previous ones.

---

Environment Variables

| Key: | Value: |
| --- | --- |
| GOROOT | /usr/local/go |
| GOPATH | //root/usr/local/go/src |
| PATH | //root/usr/local/go/src/bin:/usr/local/go... |

```
package main

import (
        "flag"
        "fmt"
        "net/http"
        "net/rpc"
)

var (
        listenAddr = flag.String("http", ":3000", "http listen address")
```

```
        dataFile   = flag.String("file", "store.json", "data store file name")
        hostname   = flag.String("host", ":3000", "http host name")
        masterAddr = flag.String("master", "", "RPC master address")
        rpcEnabled = flag.Bool("rpc", false, "enable RPC server")
)

var store Store

func main() {
        flag.Parse()
        if *masterAddr != "" {
                store = NewProxyStore(*masterAddr)
        } else {
                store = NewURLStore(*dataFile)
        }
        if *rpcEnabled { // the master is the rpc server:
                rpc.RegisterName("Store", store)
                rpc.HandleHTTP()
        }
        http.HandleFunc("/", Redirect)
        http.HandleFunc("/add", Add)
        http.ListenAndServe(*listenAddr, nil)

}

func Redirect(w http.ResponseWriter, r *http.Request) {
        key := r.URL.Path[1:]
        if key == "" {
                http.NotFound(w, r)
                return
        }
        var url string
        if err := store.Get(&key, &url); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
        http.Redirect(w, r, url, http.StatusFound)
}

func Add(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Content-Type", "text/html")
        url := r.FormValue("url")
        if url == "" {
                fmt.Fprint(w, addForm)
                return
        }
        var key string
        if err := store.Put(&url, &key); err != nil {
                http.Error(w, err.Error(), http.StatusInternalServerError)
                return
        }
        fmt.Fprintf(w, "%s", key)
}

const addForm = `
<html><body>
<form method="POST" action="/add">
URL: <input type="text" name="url">
<input type="submit" value="Add">
</form>
</html></body>
`
```

Click the **RUN** button and wait for the terminal to start. Then type `go run *.go`.

> **Remark:** To run it locally, change the port in **line 11** to `8080`. Change **line 13** as `hostname = flag.String("host","localhost:8080", "http host name")`. To test this program open browser at http://localhost:8080

---