Application for Web-IO and pure.box

pure.box Overview
To introduction video

Store Web-IO switching states in MariaDB

with Golang on the pure.box

This tutorial shows a simple Go program for the pure.box. Using the REST interface of a Web-IO its switching states are read out and stored in the MariaDB on the pure.box.

Preparation

1. Activate database on the pure.box

2. Set up database

The Maria-DB on the pure.box contains a preconfigured but empty database named **userdb** from which the device user can be accessed.

```
CREATE TABLE `iostates` (
 `id` int(11) NOT NULL,
 `time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
 `in0` int(11) NOT NULL,
 `in1` int(11) NOT NULL,
 `out0` int(11) NOT NULL,
 `out1` int(11) NOT NULL,
 `count0` int(11) NOT NULL,
 `count1` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_german2_ci;
```

To create this database scheme in the database of your purebox, use the following command:

```
mysql -u username -h box-ip -p userdb userdb.sql
```

or use a tool such as phpMyAdmin or the Mysql Workbench.

3. Install MySQL driver for Go

To access the database of the pure.box from Go, first a database driver is needed:

```
go get github.com/go-sql-driver/mysql
```

4. Activate REST interface of the Web-IO

Short paragraph

Short and sweet: REST and JSON

The Web-IO 4.0 is equipped with a REST interface which provides system information and the current states and counter states in various data formats. Since Golang inherently supports XML, JSON and Plaintext and JSON is more readable than XML at least for smaller data records, we will refer to JSON in this guide. More detailed information about REST support by the Web-IOs can be found in the operating and programming manuals.

The states of the Web-IO can be retrieved using the following URL:
`http://ADRESSE/rest/json/iostate`

The reply (without header data) is structured as follows:

```
{"iostate": {
 "input": [{
  "number": 0,
  "state": 0
 }, {
  "number": 1,
  "state": 0
 }],
 "output": [{
  "number": 0,
  "state": 0
 }, {
  "number": 1,
  "state": 0
 }],
 "counter": [{
  "number": 0,
  "state": 0
 }, {
  "number": 1,
  "state": 0
  }]
 }
}
```

The Go program

The package `encoding/json` reads in a JSON string and represents it as types. Therefore, first types are defined which represent the structure of the JSONs:

```
package main

//Result ist the root element of the JSOBN
type Result struct {
 Iostate IOState `json:"iostate"`
}

//IOState keeps collections of a device's inputs, outputs and counters
type IOState struct {
 Input   []IO `json:"input"`
 Output  []IO `json:"output"`
 Counter []IO `json:"counter"`
}

//IO keeps the number if an IO and its current state
type IO struct {
 Number int `json:"number"`
 State  int `json:"state"`
}
```

Two things must be considered here: When importing, relevant fields begin with a capital letter as per the GO convention. The JSON flags explicitly assign a JSON identifier to a field.

Reading out the Web-IO via HTTP

To read out the I/O states using REST, three packets are imported in the file header:

```
    import(
      "encoding/json"
      "io/ioutil"
      "net/http"
    )
```

encoding/json: This packet provides the routines for quickly and elegantly parsing JSON.

io/ioutil: This packet helps to write the entire data of a reader to a byte slice

net/http: This provides the necessary HTTP Get function

IP and Data URL are provided as global variables:

```
var(
 WebIOIP      string = "10.40.38.100"
 WebIODataURL string = "http://" + WebIOIP + "/rest/json/iostate"
)
```

The function getWebIOStates queries the current states using REST and returns a`Result`-type from the jsontypes.go:

```
1     func getWebIOStates() Result {
2      //get HTTP response via REST
3      response, err := http.Get(WebIODataURL)
4      if err != nil {
5       log.Fatal(err)
6      }
7      defer response.Body.Close()
8
9      //use ioutil to get json data as byte slice from response
10     jsondata, err := ioutil.ReadAll(response.Body)
11
12     //use encoding/json to unmarshal JSON
13     var result Result
14     err = json.Unmarshal(jsondata, &result)
15     if err != nil {
16      log.Fatal("Could not decode JSON!", err)
17     }
18
19     return result
20    }
```

Line 3: The states are queried via`http.Get()`.

Line 7: The keyword `defer` ensures that the response resource is enabled when ending the function.

Line 10: `ioutil.ReadAll()` returns the HTTP body as a byte slice.

Line 14: `json.Unmarshall()` parses `jsondata` and writes the result to`result`.

Linking to the database

Linking to the database requires the packet`database/sql` and the previously downloaded driver. The type of incorporation may seem somewhat confusing at first: the packet `database/sql/Driver` provides an interface. As soon as an implementation of it is imported, it can `database/sql` use this.

But since this driver is normally no longer accessed in the source code of a program, the Go compiler throws an error. The solution is the "side-effect import," with which the packet is renamed to the empty identifier '_' when importing.

```
1    import(
2      "encoding/json"
3      "io/ioutil"
4      "net/http"
5
6      "database/sql
7           _ "github.com/go-sql-driver/mysql"
8    )
```

A few parameters are also required for connecting the pure.box:

```
var (
 PureBoxIP       string = "192.168.100.25"
       PureBoxDBPort   string = "3306"
       PureBoxDBName   string = "userdb"
       PureBoxUser     string = "admin"
       PureBoxPassword string = ""

 WebIOIP      string = "192.168.100.20"
 WebIODataURL string = "http://" + WebIOIP + "/rest/json/iostate"
)
```

Now the function which writes the values to the database can be implemented:

```
1  func writeDatabase(webiodata Result) {
2
3   //Shorten variable names for the IO states
4   i0 := webiodata.Iostate.Input[0].State
5   i1 := webiodata.Iostate.Input[0].State
6   o0 := webiodata.Iostate.Output[0].State
7   o1 := webiodata.Iostate.Output[0].State
8   c0 := webiodata.Iostate.Counter[0].State
9   c1 := webiodata.Iostate.Counter[0].State
10
11  //Create dsn
12  dsn := PureBoxUser + ":" + PureBoxPassword +
13   "@" + "tcp(" + PureBoxIP + ":" + PureBoxDBPort + ")" +
14   "/" + PureBoxDBName
15
16  //Connect to database
17  db, err := sql.Open("mysql", dsn)
18  if err != nil {
19   log.Fatal("No database connection!", err)
20  }
21  defer db.Close()
22
23  //insert data
24  sqlstring := "insert into iostates (in0, in1, out0, out1, count0, count1) values(?,?,?
25  _, err = db.Query(sqlstring, i0, i1, o0, o1, c0, c1)
26  if err != nil {
27   log.Fatal("Could not write to Database! ", err)
28  }
29 }
```

Lines 4-9: Here local variables are stored an abbreviated form for receiving states.

Lines 12-14: The Data-Source-Name is used for connecting to the database. In this example it is: `admin:foo@192.168.100.25:3306/userdb`

Lines 17-19: Here the database connection is opened and closed when the function is quit.

Lines 24-28: Here follows the entry in the database. The function `db.Query()` replaces the ? in the SQL string with the transferred parameters.

The main program

For full implementation only one parameter is still missing: the time between two queries:

```
var (
 LoggingIntervall time.Duration = 10 * time.Second

 PureBoxIP       string = "192.168.100.25"
 PureBoxDBPort   string = "3306"
 PureBoxDBName   string = "userdb"
 PureBoxUser     string = "admin"
 PureBoxPassword string = "foo"

 WebIOIP      string = "192.168.100.20"
 WebIODataURL string = "http://" + WebIOIP + "/rest/json/iostate"
)
```

The main program is uncomplicated:

```
      func main() {
      for {
       iostates := getWebIOStates()
       writeDatabase(iostates)
       time.Sleep(LoggingIntervall)
       }
      }
```

Line 2: A `for` without conditions introduces an endless loop.

Line 3: In this line first the IOs are read out

Line 4: then written to the database

Line 5: and finally before another loop pass the system waits for a`LoggingIntervall`.

The complete program

As a complete source text we then have:

```
      //purebox_webio project main.go
     package main

      import (
      "log"
      "time"

      "encoding/json"
      "io/ioutil"
      "net/http"

      "database/sql"

      _ "github.com/go-sql-driver/mysql"
      )

      var (
       LoggingIntervall time.Duration = 10 * time.Second

       PureBoxIP       string = "10.40.38.10"
       PureBoxDBPort   string = "3306"
       PureBoxDBName   string = "userdb"
       PureBoxUser     string = "admin"
       PureBoxPassword string = ""

       WebIOIP      string = "10.40.38.100"
       WebIODataURL string = "http://" + WebIOIP + "/rest/json/iostate"
      )

      //Result ist the root element of the JSOBN
      type Result struct {
```

```
	Iostate IOState `json:"iostate"`
	}

	//IOState keeps collections of a device’s inputs, outputs and counters
	type IOState struct {
	Input   []IO `json:"input"`
	Output  []IO `json:"output"`
	Counter []IO `json:"counter"`
	}

	//IO keeps the number if an IO and its current state
	type IO struct {
	Number int `json:"number"`
	State  int `json:"state"`
	}

	func getWebIOStates() Result {
	//get HTTP response via REST
	response, err := http.Get(WebIODataURL)
	if err != nil {
	 log.Fatal(err)
	}
	defer response.Body.Close()

	//use ioutil to get json data as byte slice from response
	jsondata, err := ioutil.ReadAll(response.Body)

	//use encoding/json to unmarshal JSON
	var result Result
	err = json.Unmarshal(jsondata, &result)
	if err != nil {
	 log.Fatal("Could not decode JSON!", err)

	}

	return result
	}

	func writeDatabase(webiodata Result) {

	//Shorten IO names fore readability
	i0 := webiodata.Iostate.Input[0].State
	i1 := webiodata.Iostate.Input[0].State
	o0 := webiodata.Iostate.Output[0].State
	o1 := webiodata.Iostate.Output[0].State
	c0 := webiodata.Iostate.Counter[0].State
	c1 := webiodata.Iostate.Counter[0].State

	//Create dsn
	dsn := PureBoxUser + ":" + PureBoxPassword +
	 "@" + "tcp(" + PureBoxIP + ":" + PureBoxDBPort + ")" +
	 "/" + PureBoxDBName

	//Connect to database
	db, err := sql.Open("mysql", dsn)
	if err != nil {
	 log.Fatal("No database connection!", err)
	}
	defer db.Close()

	//insert data
	sqlstring := "insert into iostates (in0, in1, out0, out1, count0, count1) values(?,?,?,?,?,
	_, err = db.Query(sqlstring, i0, i1, o0, o1, c0, c1)
	if err != nil {
	 log.Fatal("Could not write to Database! ", err)
	}
	}

	func main() {
	// Never stop doing
```

```
    for {
     iostates := getWebIOStates()
     log.Println(iostates)
     writeDatabase(iostates)
     time.Sleep(LoggingIntervall)
    }
  }
```

We are available to you in person:

Wiesemann & Theis
GmbH
Porschestr. 12
42279 Wuppertal
Phone: +49 202/2680-110 (Mon.-Fri. 8 a.m. to 5 p.m.)
Fax: +49 202/2680-265
info@wut.de

Data Privacy