Porto tutorial

Thomas Hagelien

June 20, 2014

Contents

Dette er en test

1 A brief introduction to portoshell

Portoshell is a scripting tool for performing various tasks related to the NanoSim platform development, data and process management and simulator connectivity. The scripting language is based on ECMAScript, better known as JavaScript (although the language is not related to Java, and has a completely different semantics). For more background on the JavaScript language, please consult your nearest google search engine.

In addition the being ECMA-262 compatible, the Portoshell also supports custom extensions. These comes in different flavours:

- Standard build-in utility extensions (modules)
- External modules (written JavaScript)
- Plugin-in support (Written or wrapped in C++)

1.1 Standard Build in utility extensions

Add the current moment of writing, the Porto standard build-in modules are

- Process Module
- EventLoop Module
- UDP Socket Module
- Console Module
- File System Module
- File System Watcher Module
- Httpd Module

These modules are considered core featured and provide functionality to increase producitivy.

1.2 External modules

The external JavaScript modules are extensions that are easy to manipulate and reload at runtime. They are containers of data and functions that can be loaded through the build-in function:

```
require (modulename)
```

The modulename can either be a javascript file, or a modulename including namespace

```
require ('porto.mvc');  // fetching a registered module

require ('/path/to/file.js'); // fetching a module scriptfile
```

The difference between a module and a regular script file is that a module is captured within a javascript closure, accessing an object named 'exports'. Only variables and functions that are set as properties to exports will be available from the callee. This avoids naming pollution and provides nice module APIs.

```
(function (mymodule){
  mymodule.hello = function (){
    console.log ("Hello Porto Modules!");
  }
})(exports);

require('mymodule.js').hello();
// => Hello Porto Modules!
```

At the time of writing we currently have the following 'standard' external modules:

Module	Description
porto.entity	*Simplifies working with entities*
porto.collection	*Defines the API for Collections*
porto.macro	*Macro expansion module*
porto.mvc	*Model View Controller code generator framework*
porto.storage	*Storage context class with factory*
porto.storage.json	*Experimental ultralight JSON storage extension*
porto.storage.mongo	*MongoDB wrapper utility*
porto.utils.metastore	*Utility for storing metadata in the metadata database*
porto.factory.entity	*Factory for dynamic runtime generation of entity classes*

1.3 Plugin-in support (C++)

The last method of extending Portoshell is through pre-compiled plugins that are loaded at startup. These plugins can provide access to C/C++/++ libraries through C++/Scripting interfaces. Details of how this is performed is beyond the scope of this tutorial. There is currently only on plug-in available in Porto, namely the *MongoDB-plugin* which gives a low-level access to the mongodb C-driver API.

2 Creating your first script

In this section you will be introduced to scripting in portoshell. We'll start with the clichÃ© hello-world:

```
__main__ = function (args)
{
    print ("Hello, World");
}
```

To execute this script from command-line, save this script to a file *hello.js* and call

```
portoshell hello.js
```

The key idea here is that the script file will need to have defined an entry body

```
__main__ = function (args)
```

that will be called from the script engine.

If no arguments are given to portoshell, you will start the interactive REPL (Read-evaluate-print-loop). This is a useful environment for experimenting and interactive development.

```
$ portoshell
PortoShell 0.1.32
Source license: LGPLv3

For help, type :help

> print ("Hello");
```

```
Hello
undefined
>
```

Notice the text *undefined* that gets printed after the output is written. This is return value of the expression evaluation. If the expression doesn't return av valid value, the value *undefined* gets written.

3 Working with metadata

Now you have the fundamentals covered to get started working with data and metadata. Meta-data is a high-level description of data that identifies how data from any source can be interpreted. Meta-data should be uniquely indentified with a name and version number. Furthermore, the metadata should defined a list of properties that identifies names, type, units, dimensions etc for each element.

3.1 Example: Unit metadata

Let's start with a simple example: Define meta-data for the entity *unit*. The unit-entity should contain the following fields:

- abbreviation
- code
- conversionMultiplier
- conversionOffset
- quantityKind
- symbol
- unit

This follows the structure defined by QUDT
The formal meta-entity is specified in JSON like this:

```
{
    "name": "unit",
    "version": "1.0-SNAPSHOT-1",
```

```
    "description": "Unit definitions",
    "properties": [
       {
         "name": "abbreviation",
         "type": "string"
       },
       {
         "name": "code",
         "type": "string"
       },
       {
         "name": "conversionMultiplier",
         "type": "double"
       },
       {
         "name": "conversionOffset",
         "type": "double"
       },
       {
         "name": "quantityKind",
         "type": "string"
       },
       {
         "name": "symbol",
         "type": "string"
       },
       {
         "name": "unit",
         "type": "string"
       }
    ]
}
```

Save this contents in a file named 'unit.json'. To insert this file into the meta-database, perform the following operation from a shell

```
$ register-entity.sh unit.json

2014/06/14 15:00:40.0110: [18265]: INFO: cluster: Client
initialized in direct mode.
```

```
{"name":"unit","version":"1.0-SNAPSHOT-1","description":"Unit
definitions","properties":[{"name":"abbreviation","type":"string"},
{"name":"code","type":"string"},{"name":"conversionMultiplier",
"type":"double"},{"name":"conversionOffset","type":"double"},{"name":
"quantityKind","type":"string"},{"name":"symbol","type":"string"},
{"name":"unit","type":"string"}]}
```

The *register-entity.sh* script is a utility that inserts a json-file into MongoDB. If we take a look at the file *register-entity.sh*, you will find that this is not a regular shell-script at all, but a script that actually gets run by portoshell

```
#!/usr/bin/env portoshell

/*
 * register-entity.sh
 * A utility to commit meta-data into the metadata-database
 */

__main__ = function (args)
{
    if (args.length == 0) {
           console.error("fatal error: no input files");
           return undefined;
       }
    var metaStorage = require ('porto.utils.metastore').connect(
        {
            uri: 'mongodb://localhost',
            database: 'meta',
            collection: 'entities'
        });
    args.forEach(function(file){
        fs.readFile(file, function(err, data) {
            if (err) {
                print ("error:", err);
                return;
            }
            if (!metaStorage.store (data)) {
                print ("Failed to write data");
            }
```

```
        });
    });
}
```

If you look closely, you will see that our meta-database is (here) hard-coded to be the database: '/meta/', collection: '/entities/' in a mongodb server running on *localhost*. Don't worry if you don't understand the rest of the script.

The file *all-units.json* is a collection of all the documented units available from qudt.org. Our goal now is to create an instance of the entity *Unit*, fill it with the contents of the existing data, and store the data as an entity in MongoDB.

3.2 Creating instances of entities

Creating an entity of type unit can now be performed runtime in the scripting environment:

```
var entity = require('porto.entity').db(driverInfo);
Unit       = entity.using('unit', '1.0-SNAPSHOT-1');
```

Notice that we never implement the object *Unit*. This is generated for us, based on the metadata that we just stored. The return value from *entity.using* is actually the *Unit.protocol.constructor* reference. The variable '/u/' is now holding a new instance of the class '/Unit/'. It is also possible to call createEntity with a callback function that will capture errors and the class definition (the function that creates the class passed as text).

3.3 Storing data to the database

The storage where we want to store the entity values could be a number of different locations. The storage-module contains a factory pattern that allows us to specify the name of the driver we want to use, together with other info. We are not bound to a single driver such as MongoDB at this point. Any supported driver can be loaded runtime and used for storage and data retrieval.

```
/* connectivity information */
var driverInfo = {
    driver:     'mongodb',
    database:   'porto',
```

```
    collection: 'units'
};
```

In this case we want to store our data in a database called '/porto/', in a collection we call '/mydata/'. Let's fill an entity with some data and store the contents to the database:

```
var unit = new Unit();
u.set({
    abbreviation: "Gy/s",
    code: "0780",
    conversionMultiplier: "1.0e0",
    conversionOffset: "0.0",
    quantityKind: "quantity:AbsorbedDoseRate",
    symbol: "Gy/s",
    unit: "GrayPerSecond"
});
```

The *Unit* object is generated with a set of getter and setter functions. We could equally well have written something like this:

```
...
u.setAbbrevation ("Gy/s");
u.setCode ("0780");

...
```

To finalize the process, we can now store the data as a value in our database:

```
u.store();
```

The complete example should now look like this:

```
/* connectivity information */
var driverInfo = {
    driver:     'mongodb',
    database:   'porto',
    collection: 'units'
};
```

```
/* Entity creator */
var entity = require('porto.entity').db(driverInfo);
Unit       = entity.using('unit', '1.0-SNAPSHOT-1');

var u = new Unit();

u.set({
    abbreviation: "Gy/s",
    code: "0780",
    conversionMultiplier: "1.0e0",
    conversionOffset: "0.0",
    quantityKind: "quantity:AbsorbedDoseRate",
    symbol: "Gy/s",
    unit: "GrayPerSecond"
});

u.store();
print (u.id);
```

In this example we've seen how an entity can be generated runtime for us, based on meta-data. In other languages, the code generation might have to be performed compile-time, and included in the code-base. This is, however, semantically equal to what we've shown here. The key concept is that '/meta-data/' defines the schema, the storage driver is completely seperated from the implementation of the Entity instance, and the correct coupling can be performed runtime, giving extreme flexibility.

The last command in the example prints the unique identifier of the entity. This value as an universally unique identifier and needs to be communicated between processes that work with the same data.

3.4 Reading data back from a database

Our storage device have defined both read and write operations, so we can reuse our storage from the previous example. To read back data (from a different application) we can simply instanciate our entity with the UUID generated be the instance.

```
/* let's pretend the UUID we got was
8dd10147-d0b9-48ee-ae9b-2ef41d56add9 */
```

```
var id = '8dd10147-d0b9-48ee-ae9b-2ef41d56add9';
var entity = require('porto.entity').db(driverInfo);
Unit       = entity.using('unit', '1.0-SNAPSHOT-1');

var u = new Unit(id);
```

Notice that working Porto doesn't really require the developer to fight a lot of different APIs. Hiding the boilerplate code in configuration files leaves the developers to simply instanciate a class/module/object and start working with the data in a language native way.

There is another important thing to consider as well. In our client code we have said (made a contract) that we want to use the entity *unit:1.0-SNAPHOT-1*. However, we don't say anything about the source type. This is one of the more sophisticated features of the Porto design. If the datasource that stored the contents of the entity with the id= '/8dd10147-d0b9-48ee-ae9b-2ef41d56add9/', was of a different kind, the framework would notice that the client entity and source entity was different, and it would search for an explicit *translator*. The translators are simply code that accepts a given entity type and returns a different one. The client code will never have to include its own version control to accomodate for changes in file formats etc. We can simply just state *what* entity we want to use, and that's the end of that.

3.5 Introducing Collections

Collections are simply a formal specification of an entity that contains information about other entities and their relations. One of the key design principles of the Proto data centric design is the separation of data (entities) and structures (relations). The Collection constructor is defined in the external module '/porto.collection'/.

```
Collection = require('porto.collection').db(driverInfo);
var myCollection = new Collection();
```

The Collection class has the following API

Function	Description
setName(name)	Set the collection name
name()	Get the name of the collection
setVersion(version)	Set the version of the collection
version()	Get the version of the collection
count()	Return the number of registered entities
instances()	Return the label of each entity instance
findInstance(label)	Return the entity object with the given label
registerRelation(from,to,rel)	Creates a relation between two entities
registerEntity(entity,label)	Registeres an entity with a given (locally unique) label

This will create an empty collection.
Let's create a script that does the following

1. Read the complete file of units
2. Create a collection that should contain the complete set of unit entities
3. Create new entities for each unit and store the value
4. Register the entity in the collection
5. Store the collection and report the UUID:

```
var unitsJSON = fs.readFile('units.json', function(err, data){
    if (err) throw(err);

    /* connectivity information */
    var driverInfo = {
        driver:     'mongodb',
        database:   'porto',
        collection: 'units'
    };

    /* Entity and Collection creators */
    Collection = require('porto.collection').db(driverInfo);
    var entity = require('porto.entity').db(driverInfo);
    Unit       = entity.using('unit', '1.0-SNAPSHOT-1');

    /* Parse the external file into a javascript object */
    var obj = JSON.parse(data);
```

```
    /* Instanciate a new Collection class */
    var unitLibrary = new Collection();
    unitLibrary.setName('UnitCollection');
    unitLibrary.setVersion('1.0');

    /* Iterate through all the units and make a call the callback function */
    obj.units != undefined && obj.units.forEach (function(unitObj) {
        /* Create a new Entity for each element in the list */
        var unitEntity = new Unit();
        unitEntity.set(unitObj);
        unitEntity.store();

        /* Register the entity in the Collection class*/
        unitLibrary.registerEntity (unitEntity, 'entity'+unitEntity.code);
    });

    /* Store the Collection */
    unitLibrary.store();

    /* Display the id of the collection */
    console.log (unitLibrary.id);
});
```

3.6 Introducing the Porto MVC code generator

The code generator is implemented in the external module 'porto.mvc'. This module contains only one function (*create()*), which takes as the argument a model (object) and a view (template file), and returns a new function that will expand the javascript-markup contents of the view template and return a string. The generate method takes a '/bag/' object as an optional argument. Every property connected to bag will be available in the view template js-code. In addition to *bag*, the porto.model - defined in the *create(obj)* function call, will be available.

Let's create an example where we instanciate our unit library that we just stored. Then we pass the collection contents to the view template which expands the contents from the MongoDB and into a HTML-document, that will present the unit library as tables.

```
var driverInfo = {
```

```
    driver: 'mongodb',
    database: 'porto',
    collection: 'units'
};

Collection = require('soft.collection').db(driverInfo);

/* Instanciate our unitLibrary collection */
var collectionID = '79fe6b02-7b9e-4339-b238-983333b37552';
var unitLibrary  = new Collection(collectionID);

/* Create a generate function*/
var generate = require('soft.mvc').create({
    model: unitLibrary.get(),
    view: 'webtemplate.jshtml'
});

/* Store the result to an output file */
fs.writeFile ('output.html', generate(driverInfo), function(err){
    if (err) throw (err);
});
```

We will also need our template. It looks a bit ugly, but hey - it is what it is:

```
@{
  var entity = require('soft.entity').db(porto.bag);
  Unit = entity.using('unit', '1.0-SNAPSHOT-1');
  createTable = function(unit){
    return "<table style=\"width:300px\"><tr><th colspan=\"2\">unit:" +
    unit.unit + "</th></tr><tr><th>Property</th><th>Value</th></tr><tr><td>abbreviatio
    unit.abbreviation+"</td></tr><tr><td>code</td><td>"+
    unit.code+"</td></tr><tr><td>conversionMultiplier</td><td>"+
    unit.conversionMultiplier+"</td></tr><tr><td>conversion offset</td><td>"+
    unit.conversionOffset + "</td></tr><tr><td>quantityKind</td><td>"+
    unit.quantityKind+"</td></tr><tr><td>symbol</td><td>"+ unit.symbol+"</td></tr></ta
  }
}
<html>
  <head>Generated at @{new Date().toString()}
```

```
  <link rel="stylesheet" type="text/css" href="http://www.qudt.org/qudt/owl/1.0.0/styl
  </head>
  <body>
    <h1>Generated from @{porto.model.__name__} version @{porto.model.__version__}</h1>
    The original data is available from <a href="http://www.qudt.org">qudt</a>
    @{porto.model.entities.map(function(obj){
      var unit = new Unit(obj.uid);
      return createTable(unit);
    }).join("<br/>");}
  </body>
</html>
```

Run the script and enjoy the results.

3.7 Summary

In this chapter we've touched upon some key features of Porto:1

- Defining meta-data schemas and storing them in a database
- Creating instances of entities that are bound the meta-data schema
- Creating collections of entities
- Storing and retreiving data using the generic storage interface
- Generating code based on data contents stored in the MongoDB database

4 Working with OpenFOAM

In this chapter we show how Porto can be used to generate inputs for OpenFOAM. This is a partial example and should be extended to a complete simulation environment run from within Porto.

4.1 Defining meta-data

As always, we start with the data modelling and implementing our meta-data schemas. The simplest OpenFOAM entity to define is proably the OpenFOAM control dictionary *controlDict*

We could be more explicit in the definition and give the properties units etc. It would also be better to use enumerators instead of string-types for some of the properties.

```
{
    "name"        : "controlDict",
    "version"     : "0.1",
    "description" : "Time and data input/output control",
    "properties"  : [
        {
            "name": "application",
            "type": "string"
        },
        {
            "name": "startFrom",
            "type": "string"
        },
        {
            "name": "startTime",
            "type": "double"
        },
        {
            "name": "stopAt",
            "type": "string"
        },
        {
            "name": "endTime",
            "type": "double"
        },
        {
            "name": "deltaT",
            "type": "double"
        },
        {
            "name": "writeControl",
            "type": "string"
        },
        {
            "name": "writeInterval",
            "type": "double"
        },
        {
            "name": "purgeWrite",
            "type": "integer"
```

```
        },
        {
            "name": "writeFormat",
            "type": "string"
        },
        {
            "name": "writePrecision",
            "type": "integer"
        },
        {
            "name": "writeCompression",
            "type": "string"
        },
        {
            "name": "timeFormat",
            "type": "string"
        },
        {
            "name": "timePrecision",
            "type": "integer"
        },
        {
            "name": "runTimeModifiable",
            "type": "string"
        },
        {
            "name": "adjustTimeStep",
            "type": "string"
        }
    ]
}
```

Let's save this file under the name '/controldict.json/'

4.2 Storing the meta-data

The next step is to store this data to the meta-data database. We can do this by using the utility *register-entity.sh*.

```
$ register-entity.sh controldict.json
```

```
2014/06/20 15:08:27.0408: [16726]: INFO: cluster: Client initialized
in direct mode.
{"name":"controlDict","version":"0.1","description":"Time and data
input/output
control","properties":[{"name":"application","type":"string"},
{"name":"startFrom","type":"string"},{"name":"startTime","type":"double"},
{"name":"stopAt","type":"string"},{"name":"endTime","type":"double"},{"name":
"deltaT","type":"double"},{"name":"writeControl","type":"string"},{"name":
"writeInterval","type":"double"},{"name":"purgeWrite","type":"integer"},
{"name":"writeFormat","type":"string"},{"name":"writePrecision","type":"integer"},
{"name":"writeCompression","type":"string"},{"name":"timeFormat","type":"string"},
{"name":"timePrecision","type":"integer"},{"name":"runTimeModifiable","type":"string"}
{"name":"adjustTimeStep","type":"string"}]}
```

4.3 Instanciate an ControlDict entity

Now that we have the meta-data available, we can instanciate the ControlDict object and give it some data:

```
var entity = require('soft.entity').db({
  driver:     'mongodb',
  database:   'porto',
  collection: 'openfoam'
});

ControlDict = entity.using('controlDict', '0.1');

/* Create an instance and give it some data*/
var nozzleControlDict = new ControlDict();
nozzleControlDict.set({
  application       : 'mdFoam',
  startFrom         : 'startTime',
  startTime         : 0,
  stopAt            : 'endTime',
  endTime           : 2e-13,
  deltaT            : 1e-15,
  writeControl      : 'runTime',
  writeInterval     : 5e-14,
  purgeWrite        : 0,
```

```
  writeFormat       : 'ascii',
  writePrecision    : 12,
  writeCompression  : 'off',
  timeFormat        : 'general',
  timePrecision     : '6',
  runTimeModifiable : 'true',
  adjustTimeStep    : 'no'
});

/* Store the entity in the database */
nozzleControlDict.store();

/* Display the UUID for further use */
print (nozzleControlDict.id);
```

4.4 Create the template view for the generated file

The next step is to make a template view for the file to be generated.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.1.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
|-------------------------------------------------------------------------------|
    Generated by Porto @{new Date().toString()}
\*-------------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

application     @{porto.model.application};

startFrom       @{porto.model.startFrom};
```

```
startTime       @{porto.model.startTime};

stopAt          @{porto.model.stopAt};

endTime         @{porto.model.endTime};

deltaT          @{porto.model.deltaT};

writeControl    @{porto.model.writeControl};

writeInterval   @{porto.model.writeInterval};

purgeWrite      @{porto.model.purgeWrite};

writeFormat     @{porto.model.writeFormat};

writePrecision  @{porto.model.writePrecision};

writeCompression @{porto.model.writeCompression};

timeFormat      @{porto.model.timeFormat};

timePrecision   @{porto.model.timePrecision};

runTimeModifiable @{porto.model.runTimeModifiable};

adjustTimeStep  @{porto.model.adjustTimeStep};

// ************************************************************************* //
```

4.5 Generate a controlDict file

The last step is to generate the code that makes the controlDict.

```
var entity = require('soft.entity').db({
  driver:     'mongodb',
  database:   'porto',
  collection: 'openfoam'});
```

```
ControlDict = entity.using('controlDict', '0.1');

var controlDict = new ControlDict('e12686f9-b677-49dc-ad9d-07944f9b053e');
var generate = require('soft.mvc').create({
  model: controlDict.get(),
  view : 'controlDict.foamjs'});

fs.writeFile('controlDict', generate(), function(err){
  if (err) throw (err);
});
```

Running this script will create a completely heatly OpenFOAM controlDict dictionary file.

5 Using UDP to communicate and run external processes

The Porto framework is not limited to just storing and retrieving data. It can also be utilized for controlling the process workflows. One syncronization mechanism that is supported is User Datagram Protocol. This a lightweight protocol with no handshaking and setup. UDP works by emitting datagrams. The datagram can be any text. In our example code, we want to create to script applications. One is the client-code that emits a JSON datagram that contains some info, along with a program w/arguments that it wants the receiver to run.

The other is the server code. This is a scipt application that creates a callback that is called when it receives a datagram. It will then parse the datagram and execute the application. Note that this method of calling remote procedure is **not** recommended due to the security hazard this exposes.

UdpSocket is a build-in utility class, and does not require us to call *require*.

```
__main__ = function (args)
{
  var udpSocket = new UdpSocket();
  var msg = {
      name: "test",
```

```
      version: "1.0",
      program: "ls",
      args: ["-al"]
  };
  udpSocket.writeDatagram(JSON.stringify (msg), "127.0.0.1", 1234);
}
```

When executed, this script will simply emit the datagram and exit. The server code is a bit more elaborate and contains some features of Porto that has not yet been discussed. The server is a script that should set up an event loop. The event loop makes it possible to create asyncrounous callbacks that is build in the Qt signal/slot mechanism, and is a build-in utility class called *EventLoop*.

In addition we also need to use the utility class *Process* which can control the execution of an application, with asyncronous callback to catch output coming from stdout, stderr, along with messages giving the status of the running application, and a callback for the termination of the application.

```
__main__ = function (args)
{
    /* Create the event loop */
    var event = new EventLoop ();

    var udpSocket = require ('./udpsocket.js');
    var u = udpSocket.create (function (udp) {
        /* Create a callback for the event 'readyRead' */
        udp.readyRead.connect (function () {

            /* Parse the datagram received and print its contents to the console */
            var datagram = udp.readDatagram ();
            var o = JSON.parse(datagram.datagram);
            print(o.name, o.version, o.program);

            /* Instanciate a new Process and create callbacks for when the process
               starts, have available standard output contents, and termination.
             */
            var proc = new Process();
            (function (p) {
                p.started.connect (function (){
                    print ("process started");
```

```
                });

                p.readyReadStandardOutput.connect (function() {
                    print (p.readAllStandardOutput ());
                });

                p["finished(int)"].connect (function (){
                    print ("process finished");
                    event.quit(); /* Quit the server when the program finishes*/
                });
            })(proc);

            /* Set up the process with the program and arguments given in the datagram
            proc.setProgram(o.program);
            proc.setArguments(o.args);

            /* Run the application */
            proc.start();
        });
    });

    /* Make the socket listen for activity on port 1234 coming from localhost */
    u.bind ("127.0.0.1", 1234);

    /* Enter the event loop */
    event.exec();
}
```

The module udpsocket.js is just a simple high-level function that instanciates the UdpSocket and pass it to a function argument.

```
exports.create = function(fn) {
    var udp = new UdpSocket();
    fn (udp);

    return udp;
}
```

6 Displaying contents in a web browser

Working in a scripting shell environment is sometimes not practical when it comes to presenting information. Creating or generating HTML for presenting contents is therefore good options. One could also consider to build *Porto User Interfaces* as web services. This chapter will demonstrate the functionality of the tiny webserver functionality available in *Porto*, in the build-in utility HttpServer.

```
__main__ = function (args)
{
  var event = new EventLoop();
  var port  = 8081;
  var httpd = new HttpServer(port);
  httpd.setRootDir("html/");
  httpd.start();
  event.exec();
}
```

By pointing the browser to http://localhost:8081 we should now be able to view the contents defined under the given root directory.

Note hoewever, that is not a production web server, and it is limited to GET requests of type text/html.

7 Final notes

This tutorial/overview was intended to give the reader some hands-on to get started on working with Porto. There are many more things that Porto is able to do in terms of features, but the key concepts are the most important. I wish this document has left the reader with a little deeper understanding of the data-centric design philosophy, and also inspiration to contribute and extend the framework to fit other tasks and activities. After all, software is supposed to make work easier, more fun, and inspire new ideas.

Also note that Porto is in a **very early stage of development**. Do not expect everything to be smooth and easy, but please make a note of annoyances, problems, improvement suggestions, and feel free to make these requirements for the development interation.

Thomas Hagelien Trondheim 2014