Embark Documentation

Release 2.4.0

Iuri Matias <>

Apr 04, 2017

Contents

This is a work in progress, feel free to contribute!

CHAPTER 1

Installation

Requirements: geth (1.5.8 or higher), node (6.9.1 or higher is recommended) and npm serpent (develop) if using contracts with Serpent, testrpc (3.0 or higher) if using the simulator or the test functionality. Further: depending on the dapp stack you choose: IPFS

```
$ npm -g install embark

# If you plan to use the simulator instead of a real ethereum node.
$ npm -g install ethereumjs-testrpc
```

See Complete Installation Instructions.

updating from embark 1

Embark's npm package has changed from `embark-framework` to `embark`, this sometimes can create conflicts. To update first uninstall embark-framework 1 to avoid any conflicts. `npm uninstall -g embark-framework` then `npm install -g embark`

CHAPTER 2

Usage

CHAPTER 3

Usage - Demo

You can easily create a sample working DApp with the following:

```
$ embark demo
$ cd embark_demo
```

You can run a REAL ethereum node for development purposes:

```
$ embark blockchain
```

Alternatively, to use an ethereum rpc simulator simply run:

```
$ embark simulator
```

By default embark blockchain will mine a minimum amount of ether and will only mine when new transactions come in. This is quite usefull to keep a low CPU. The option can be configured at `config/blockchain.json`. Note that running a real node requires at least 2GB of free ram, please take this into account if running it in a VM.

Then, in another command line:

```
$ embark run
```

This will automatically deploy the contracts, update their JS bindings and deploy your DApp to a local server at http://localhost:8000

Note that if you update your code it will automatically be re-deployed, contracts included. There is no need to restart embark, refreshing the page on the browser will do.

CHAPTER 4

Dashboard

Embark 2 comes with a terminal dashboard.

Fig. 4.1: Dashboard

The dashboard will tell you the state of your contracts, the enviroment you are using, and what embark is doing at the moment.

available services

Available Services will display the services available to your dapp in green, if one of these is down then it will be displayed in red.

logs and console

There is a console at the bottom which can be used to interact with contracts or with embark itself. type `help` to see a list of available commands, more commands will be added with each version of Embark.

CHAPTER 5

Creating a new DApp

If you want to create a blank new app.

```
$ embark new AppName
$ cd AppName
```

To run Embark then in one console run:

```
$ embark blockchain
```

And in another console run:

```
$ embark run
```

CHAPTER 6

DApp Structure

```
app/
  |___ contracts/ #solidity smart contracts
  |___ html/
  |___ css/
  |___ js/
config/
  |___ blockchain.json #rpc and blockchain configuration
  |___ contracts.json  #ethereum contracts configuration
  |___ storage.json  #ipfs configuration
  |___ communication.json  #whisper/orbit configuration
  |___ webserver.json  #dev webserver configuration
test/
  |___ #contracts tests
```

Solidity files in the contracts directory will automatically be deployed with `embark run`. Changes in any files will automatically be reflected in app, changes to contracts will result in a redeployment and update of their JS Bindings

CHAPTER 7

Libraries and languages available

Embark can build and deploy contracts coded in Solidity or Serpent. It will make them available on the client side using EmbarkJS and Web3.js.

Further documentation for these can be found below:

- Smart Contracts: Solidity and Serpent
- Client Side: Web3.js and *EmbarkJS*

CHAPTER 8

Configuring & Using Contracts

Embark will automatically take care of deployment for you and set all needed JS bindings. For example, the contract below:

```
# app/contracts/simple_storage.sol
contract SimpleStorage {
  uint public storedData;

  function SimpleStorage(uint initialValue) {
    storedData = initialValue;
  }

  function set(uint x) {
    storedData = x;
  }
  function get() constant returns (uint retVal) {
    return storedData;
  }
}
```

Will automatically be available in Javascript as:

```
# app/js/index.js
SimpleStorage.set(100);
SimpleStorage.get().then(function(value) { console.log(value.toNumber()) });
SimpleStorage.storedData().then(function(value) { console.log(value.toNumber()) });
```

You can specify for each contract and environment its gas costs and arguments:

```
# config/contracts.json
{
  "development": {
    "gas": "auto",
    "contracts": {
      "SimpleStorage": {
        "args": [
```

```
          100
        ]
      }
    }
  }
}
```

If you are using multiple contracts, you can pass a reference to another contract as `$ContractName`, Embark will automatically replace this with the correct address for the contract.

```
# config/contracts.json
{
  ...
  "development": {
    "contracts": {
      "SimpleStorage": {
        "args": [
          100,
          $MyStorage
        ]
      },
      "MyStorage": {
        "args": [
          "initial string"
        ]
      },
      "MyMainContract": {
        "args": [
          $SimpleStorage
        ]
      }
    }
  }
  ...
}
```

You can now deploy many instances of the same contract. e.g

```
# config/contracts.json
{
  "development": {
    "contracts": {
      "Currency": {
        "deploy": false,
        "args": [
          100
        ]
      },
      "Usd": {
        "instanceOf": "Currency",
        "args": [
          200
        ]
      },
      "MyCoin": {
        "instanceOf": "Currency",
        "args": [
          200
```

```
        ]
      }
    }
  }
}
  ...
```

Contracts addresses can be defined, If an address is defined the contract wouldn't be deployed but its defined address will be used instead.

```
# config/contracts.json
{
  ...
  "development": {
    "contracts": {
      "UserStorage": {
        "address": "0x123456"
      },
      "UserManagement": {
        "args": [
          "$UserStorage"
        ]
      }
    }
  }
  ...
}
```

CHAPTER 9

Configuring Storage (IPFS)

Embark will check your prefered storage configuration in the file `config/storage.json`. This file will contain the prefered configuration for each environment. With `default` being the configuration fields that applies to every environment. Each of those can be individually overriden in a per environment basis.

e.g :

```
{
  "default": {
    "enabled": true,
    "ipfs_bin": "ipfs",
    "provider": "ipfs",
    "available_providers": ["ipfs"],
    "host": "localhost",
    "port": 5001
  },
  "development": {
    "enabled": true,
    "provider": "ipfs",
    "host": "localhost",
    "port": 5001
  }
}
```

options available:

- `enabled` (boolean: true/false) to enable or completly disable storage support
- `ipfs_bin` (string) name or desired path to the ipfs binary
- `provider` (string: "ipfs") desired provider to automatically connect to on the dapp. e.g in the example above, seting this to `"ipfs"` will automaticaly add `EmbarkJS.setProvider('ipfs', {server: 'localhost', 5001})` to the generated code
- `available_providers` (array: ["ipfs"]) list of storages to be supported on the dapp. This will affect what's available with the EmbarkJS library on the dapp.
- `host` and `port` of the ipfs node to connect to.

CHAPTER 10

Configuring Communication (Whisper, Orbit)

Embark will check your prefered communication configuration in the file `config/communication.json`. This file will contain the prefered configuration for each environment. With `default` being the configuration fields that applies to every environment. Each of those can be individually overriden in a per environment basis.

e.g :

```
{
  "default": {
    "enabled": true,
    "provider": "whisper",
    "available_providers": ["whisper", "orbit"]
  }
}
```

options available:

- `enabled` (boolean: true/false) to enable or completly disable communication support
- `provider` (string: "wisper" or "orbit") desired provider to automatically connect to on the dapp. e.g in the example above, seting this to `"whisper"` will automaticaly add `EmbarkJS.setProvider('whisper')` to the generated code
- `available_providers` (array: ["whisper", "orbit"]) list of communication platforms to be supported on the dapp. This will affect what's available with the EmbarkJS library on the dapp so if you don't need Orbit for e.g, removing it from this will considerably reduce the file size of the generated JS code.

CHAPTER 11

EmbarkJS

EmbarkJS is a javascript library meant to abstract and facilitate the development of DApps.

promises

methods in EmbarkJS contracts will be converted to promises.

```
var myContract = new EmbarkJS.Contract({abi: abiObject, address: "0x123"});
myContract.get().then(function(value) { console.log("value is " + value.toNumber) });
```

deployment

Client side deployment will be automatically available in Embark for existing contracts:

```
SimpleStorage.deploy().then(function(anotherSimpleStorage) {});
```

or it can be manually definied as

```
var myContract = new EmbarkJS.Contract({abi: abiObject, code: code});
myContract.deploy().then(function(anotherMyContractObject) {});
```

CHAPTER 12

EmbarkJS - Storage (IPFS)

initialization

The current available storage is IPFS. it can be initialized as

```
EmbarkJS.Storage.setProvider('ipfs',{server: 'localhost', port: '5001'})
```

Saving Text

```
EmbarkJS.Storage.saveText("hello world").then(function(hash) {});
```

Retrieving Data/Text

```
EmbarkJS.Storage.get(hash).then(function(content) {});
```

Uploading a file

```
<input type="file">
```

```
var input = $("input[type=file"]);
EmbarkJS.Storage.uploadFile(input).then(function(hash) {});
```

Generate URL to file

```
EmbarkJS.Storage.getUrl(hash);
```

CHAPTER 13

EmbarkJS - Communication (Whisper, Orbit)

initialization

For Whisper:

```
EmbarkJS.Messages.setProvider('whisper')
```

For Orbit:

You'll need to use IPFS from master and run it as: `ipfs daemon --enable-pubsub-experiment`

then set the provider:

```
EmbarkJS.Messages.setProvider('orbit', {server: 'localhost', port: 5001})
```

listening to messages

```
EmbarkJS.Messages.listenTo({topic: ["topic1", "topic2"]}).then(function(message) {␣
↪console.log("received: " + message); })
```

sending messages

you can send plain text

```
EmbarkJS.Messages.sendMessage({topic: "sometopic", data: 'hello world'})
```

or an object

```
EmbarkJS.Messages.sendMessage({topic: "sometopic", data: {msg: 'hello world'}})
```

note: array of topics are considered an AND. In Whisper you can use another array for OR combinations of several topics e.g `["topic1", ["topic2", "topic3"]]` => `topic1 AND (topic2 OR topic 3)`

CHAPTER 14

Testing Ethereum Contracts

You can run specs with `embark test`, it will run any test files under `test/`.

Embark includes a testing lib to fastly run & test your contracts in a EVM.

```
# test/simple_storage_spec.js

var assert = require('assert');
var Embark = require('embark');
var EmbarkSpec = Embark.initTests();
var web3 = EmbarkSpec.web3;

describe("SimpleStorage", function() {
  before(function(done) {
    var contractsConfig = {
      "SimpleStorage": {
        args: [100]
      }
    };
    EmbarkSpec.deployAll(contractsConfig, done);
  });

  it("should set constructor value", function(done) {
    SimpleStorage.storedData(function(err, result) {
      assert.equal(result.toNumber(), 100);
      done();
    });
  });

  it("set storage value", function(done) {
    SimpleStorage.set(150, function() {
      SimpleStorage.get(function(err, result) {
        assert.equal(result.toNumber(), 150);
        done();
      });
    });
```

```
  });

});
```

Embark uses Mocha by default, but you can use any testing framework you want.

CHAPTER 15

Working with different chains

You can specify which environment to deploy to:

```
$ embark blockchain livenet
```

```
$ embark run livenet
```

The environment is a specific blockchain configuration that can be managed at config/blockchain.json

```
# config/blockchain.json
  ...
   "livenet": {
    "networkType": "livenet",
    "rpcHost": "localhost",
    "rpcPort": 8545,
    "rpcCorsDomain": "http://localhost:8000",
    "account": {
      "password": "config/livenet/password"
    }
  },
  ...
```

CHAPTER 16

Structuring Application

Embark is quite flexible and you can configure you're own directory structure using `embark.json`

```
# embark.json
{
  "contracts": ["app/contracts/**"],
  "app": {
    "css/app.css": ["app/css/**"],
    "images/": ["app/images/**"],
    "js/app.js": ["embark.js", "app/js/**"],
    "index.html": "app/index.html"
  },
  "buildDir": "dist/",
  "config": "config/",
  "plugins": {}
}
```

CHAPTER 17

Deploying to IPFS

To deploy a dapp to IPFS, all you need to do is run a local IPFS node and then run `embark upload ipfs`. If you want to deploy to the livenet then after configuring you account on `config/blockchain.json` on the `livenet` environment then you can deploy to that chain by specifying the environment `embark ipfs livenet`.

CHAPTER 18

Deploying to SWARM

To deploy a dapp to SWARM, all you need to do is run a local SWARM node and then run `embark upload swarm`.

CHAPTER 19

Extending functionality with plugins

To add a plugin to embark:

1. Add the npm package to package.json e.g `npm install embark-babel --save`
2. Then add the package to `plugins:` in embark.json e.g `"plugins": { "embark-babel": {} }`

Creating a plugin:

1. `mkdir yourpluginname`
2. `cd yourpluginname`
3. `npm init`
4. create and edit `index.js`
5. add the following code:

```
module.exports = function(embark) {
}
```

The `embark` object then provides an api to extend different functionality of embark.

Usecases examples

- plugin to add support for es6, jsx, coffescript, etc (`embark.registerPipeline`)
- plugin to add standard contracts or a contract framework (`embark.registerContractConfiguration` and `embark.addContractFile`)
- plugin to make some contracts available in all environments for use by other contracts or the dapp itself e.g a Token, a DAO, ENS, etc.. (`embark.registerContractConfiguration` and `embark.addContractFile`)
- plugin to add a libraries such as react or boostrap (`embark.addFileToPipeline`)
- plugin to specify a particular web3 initialization for special provider uses (`embark.registerClientWeb3Provider`)
- plugin to create a different contract wrapper (`embark.registerContractsGeneration`)

- plugin to add new console commands (`embark.registerConsoleCommand`)
- plugin to add support for another contract language such as viper, LLL, etc (`embark.registerCompiler`)
- plugin that executes certain actions when contracts are deployed (`embark.events.on`)

embark.pluginConfig

Object containing the config for the plugin specified in embark.json, for e.g with:

```
"plugins": {
  "embark-babel": { "files": ["**/*.js", "!**/jquery.min.js"], "presets": ["es2015",
↪"react"] }
}
```

`embark.pluginConfig` will contain `{ "files": ["**/*.js", "!**/jquery.min.js"], "presets": ["es2015", "react"] }`

embark.registerPipeline(matchingFiles, callback(options))

This call will return the content of the current asset file so the plugin can transform it in some way. Typically this is used to implement pipeline plugins such as Babel, JSX, sass to css, etc..

`matchingFiles` is an array of matching files the plugin should be called for e.g [`**/*.js`, `!vendor/jquery.js`] matches all javascript files except vendor/jquery.js

options available:

- targetFile - filename to be generated
- source - content of the file

expected return: `string`

```
var babel = require("babel-core");
require("babel-preset-react");

module.exports = function(embark) {
    embark.registerPipeline(["**/*.js", "**/*.jsx"], function(options) {
      return babel.transform(options.source, {minified: true, presets: ['react']}).
↪code;
    });
}
```

embark.registerContractConfiguration(contractsConfig)

This call is used to specify a configure of one or more contracts in one or several environments. This is useful for specifying the different configurations a contract might have depending on the enviroment. For instance in the code bellow, the `DGDToken` contract code will redeployed with the arguments `100` in any environment, except for the livenet since it's already deployed there at a particular address.

Typically this call is used in combination with `embark.addContractFile`

`contractsConfig` is an object in the same structure as the one found in the contracts configuration at `config/contracts.json`. The users own configuration will be merged with the one specified in the plugins.

```
module.exports = function(embark) {
    embark.registerContractConfiguration({
      "default": {
        "contracts": {
          "DGDToken": {
            "args": [
              100
```

```
            ]
          }
        }
      },
      "livenet": {
        "contracts": {
          "DGDToken": {
            "address": "0xe0b7927c4af23765cb51314a0e0521a9645f0e2a"
          }
        }
      }
    });
}
```

embark.addContractFile(file)

Typically this call is used in combination with `embark.registerContractConfiguration`. If you want to make the contract available but not automatically deployed without the user specifying so you can use `registerContractConfiguration` to set the contract config to `deploy: false`, this is particularly useful for when the user is meant to extend the contract being given (e.g `contract MyToken is StandardToken`)

`file` is the contract file to add to embark, the path should relative to the plugin.

```
module.exports = function(embark) {
    embark.addContractFile("./DGDToken.sol");
}
```

embark.addFileToPipeline(file, options)

This call is used to add a file to the pipeline so it's included with the dapp on the client side.

`file` is the file to add to the pipeline, the path should relative to the plugin.

`options` available:

- skipPipeline - If true it will not apply transformations to the file. For example if you have a babel plugin to transform es6 code or a minifier plugin, setting this to true will not apply the plugin on this file.

```
module.exports = function(embark) {
    embark.addFileToPipeline("./jquery.js", {skipPipeline: true});
}
```

embark.registerClientWeb3Provider(callback(options))

This call can be used to override the default web3 object generation in the dapp. it's useful if you want to add a plugin to interact with services like http://infura.io or if you want to use your own web3.js library extension.

options available:

- rpcHost - configured rpc Host to connect to
- rpcPort - configured rpc Port to connect to
- blockchainConfig - object containing the full blockchain configuration for the current environment

expected return: `string`

example:

```
module.exports = function(embark) {
    embark.registerClientWeb3Provider(function(options) {
        return "web3 = new Web3(new Web3.providers.HttpProvider('http://" + options.
↪rpcHost + ":" + options.rpcPort + "'));";
```

```
    });
}
```

embark.registerContractsGeneration(callback(options))

By default Embark will use EmbarkJS to declare contracts in the dapp. You can override and use your own client side library.

options available:

- contracts - Hash of objects containing all the deployed contracts. (key: contractName, value: contract object)
- abiDefinition
- code
- deployedAddress
- gasEstimates
- gas
- gasPrice
- runtimeByteCode

expected return: `string`

```
module.exports = function(embark) {
    embark.registerContractsGeneration(function(options) {
      for(var className in this.contractsManager.contracts) {
        var abi = JSON.stringify(contract.abiDefinition);

        return className + " = " + web3.eth.contract(" + abi + ").at('" + contract.
↪deployedAddress + "');";
      }
    });
}
```

embark.registerConsoleCommand(callback(options))

This call is used to extend the console with custom commands.

expected return: `string` (output to print in console) or `boolean` (skip command if false)

```
module.exports = function(embark) {
    embark.registerConsoleCommand(function(cmd, options) {
      if (cmd === "hello") {
        return "hello there!";
      }
      // continue to embark or next plugin;
      return false;
    });
}
```

embark.registerCompiler(extension, callback(contractFiles, doneCallback))

expected doneCallback arguments: `err` and `hash` of compiled contracts

- Hash of objects containing the compiled contracts. (key: contractName, value: contract object)
  - code - contract bytecode (string)

- runtimeBytecode - contract runtimeBytecode (string)
- gasEstimates - gas estimates for constructor and methods (hash)
- e.g `{"creation":[20131,38200],"external":{"get()":269, "set(uint256)":20163,"storedData()":224},"internal":{}}`
- functionHashes - object with methods and their corresponding hash identifier (hash)
- e.g `{"get()":"6d4ce63c","set(uint256)":"60fe47b1", "storedData()":"2a1afcd9"}`
- abiDefinition - contract abi (array of objects)
- e.g `[{"constant":true,"inputs":[],"name":"storedData", "outputs":[{"name":"","type":"uint256"}],"payable":false, "type":"function"}, etc...`

below a possible implementation of a solcjs plugin:

```
var solc = require('solc');

module.exports = function(embark) {
    embark.registerCompiler(".sol", function(contractFiles, cb) {
      // prepare input for solc
      var input = {};
      for (var i = 0; i < contractFiles.length; i++) {
        var filename = contractFiles[i].filename.replace('app/contracts/','');
        input[filename] = contractFiles[i].content.toString();
      }

      // compile files
      var output = solc.compile({sources: input}, 1);

      // generate the compileObject expected by embark
      var json = output.contracts;
      var compiled_object = {};
      for (var className in json) {
        var contract = json[className];

        compiled_object[className] = {};
        compiled_object[className].code            = contract.bytecode;
        compiled_object[className].runtimeBytecode = contract.runtimeBytecode;
        compiled_object[className].gasEstimates    = contract.gasEstimates;
        compiled_object[className].functionHashes  = contract.functionHashes;
        compiled_object[className].abiDefinition   = JSON.parse(contract.interface);
      }

      cb(null, compiled_object);
    });
}
```

embark.logger

To print messages to the embark log is it better to use `embark.logger` instead of `console`.

e.g `embark.logger.info("hello")`

embark.events.on(eventName, callback(*args))

This call is used to listen and react to events that happen in Embark such as contract deployment

- eventName - name of event to listen to * available events:

- "contractsDeployed" - triggered when contracts have been deployed
- "file-add", "file-change", "file-remove", "file-event" - triggered on a file change, args is (filetype, path)
- "abi", "abi-vanila", "abi-contracts-vanila" - triggered when contracts have been deployed and returns the generated JS code
- "outputDone" - triggered when dapp is (re)generated
- "firstDeploymentDone" - triggered when the dapp is deployed and generated for the first time

```
module.exports = function(embark) {
    embark.events.on("contractsDeployed", function() {
      embark.logger.info("plugin says: your contracts have been deployed");
    });
    embark.events.on("file-changed", function(filetype, path) {
      if (type === 'contract') {
        embark.logger.info("plugin says: you just changed the contract at " + path);
      }
    });
}
```

CHAPTER 20

Using Embark with Grunt

1. Edit embark.json

Edit `embark.json` to have the line `"js/app.js": ["embark.js"]`, this will make embark create the file containing the contracts initilization to `dist/app.js`.

```
{
  "contracts": ["app/contracts/**"],
  "app": {
    "app.js": ["embark.js"]
  },
  "buildDir": "dist/",
  "config": "config/",
  "plugins": {
  }
}
```

2. add the generated file to Grunt config file so it's included with the other assets

```
module.exports = (grunt) ->

  grunt.initConfig(
    files:
      js:
        src: [
          "dist/app.js"
          "app/js/**/*.js"
        ]
```

CHAPTER 21

Donations

If you like Embark please consider donating to 0x8811FdF0F988f0CD1B7E9DE252ABfA5b18c1cDb1

CHAPTER 22

Indices and tables

- genindex
- modindex
- search