EthereumJS Documentation

Release 0.1

EthereumJS Team

Dec 18, 2018

Contents:

This guide aims to be a both comprehensive and lightweight guide to the EthereumJS ecosystem. It is meant to serve as an internal reference, give guidance for new contributors and overall provide transparency on current and future work and standards and best practices applied.

CHAPTER 1

Introduction

1.1 Overview

`EthereumJS` is the **JavaScript / TypeScript** project within the **Ethereum Foundation**. Work is mainly done within the following GitHub organization:

- https://github.com/ethereumjs

There are currently over 20 active repositories, most (with some exceptions like keythereum or ethrpc) managed by the EF team. Some central libraries are an implementation of the Ethereum Virtual Machine, our Client project and implementations of the Merkle Patricia Tree data structure or the devp2p networking stack.

Have a look at the overview page linked above to get an impression what is currently being worked on as well as other libraries available.

1.2 Focus and related Projects

Main focus of `EthereumJS` is to provide high-quality and robust implementations of *core Ethereum infrastructure technologies* (virtual machine), *protocols* (devp2p) and *data structures* (merkle tree).

Other related projects you might want to check out as well are e.g.:

- web3.js (Ethereum JavaScript API)
- ethers.js (Ethereum Wallet implementation and library)
- Truffle (Development Framework)
- embark (dApp Framework)
- Remix (https://github.com/ethereum/remix)

Most of the projects above also make use of some of our base-layer libraries. `EthereumJS` libraries are also used by various other actors within the ecosystem like MetaMask, 0x or Augur.

1.3 Team and Contact

EthereumJS is a strongly community-driven project and the active team is regarded as the sum of people actively contributing to the libraries. Some people are also hired by the Ethereum Foundation to provide a somewhat more solid ground on ongoing development and having people with in-depth knowledge as steady contact persons for the libraries.

For technical questions as well as getting in touch you can use our Gitter channel:

- https://gitter.im/ethereum/ethereumjs

Organizational questions are centered and discussed on the organization repo:

- https://github.com/ethereumjs/organization

1.4 Ongoing Work Tasks

The following is an overview on ongoing work tasks to get an idea on the current focus of work. This is also serving internal accounting purposes.

Note: This list is focussing on reoccuring work tasks, for an overview on dedicated new projects have a look at the *Roadmap* section.

1.4.1 W1 - Virtual Machine Development

One strong emphasis of EthereumJS work is on maintaining and further developing a robust and up-to-date JavaScript virtual machine implementation (ethereumjs-vm).

Main tasks around this are:

- Updating the VM on new hardforks
- Targeting compliance with the latest consensus test suite releases
- Implementing feature requests from the community (Truffle, Remix, others), e.g. to provide better debugging functionalities
- Ongoing refactoring work to open up new use cases

1.4.2 W2 - Library Modernization

EthereumJS libraries provide robust and solid implementations surving the dedicated purposes, but code and API on many libraries lacks a bit behind regarding modern, well-to-read and easy to use JavaScript features.

There is an ongoing effort to modernize the libraries, this nevertheless takes time to not introduce new bugs or break existing APIs.

Currently there is a focus on (you are very much invited to help :-)):

- Using ES6 classes for structuring library components
- JavaScript Promise based interfaces (in contrast to callback logic)
- Updating on security improving language features (block-scoped variables,...)
- Improving on code readability (destructuring of objects,...)

Note: Complementary to this work is the task of establishing a robust transpilation process with tools like `babel` and others to make sure that our libraries are usable in environments where modern JS language features are not yet available

1.4.3 W3 - Bug Fixes and Maintenance

`EthereumJS` libraries are widely used in production - often in security-sensitive contexts - and there is an ongoing effort to keep libraries up-to-date and secure.

Main tasks around this:

- Fix bugs reported by the community in a timely fashion
- Keep library dependencies up-to-date
- Adopt libraries to various user work environments and build pipelines (browser, React,...)
- Be responsive to feature requests from the community

1.4.4 W4 - Testing and CI

To provide a high level of reliable we target a high test coverage on all of our libraries and writing new tests and integrate these in the everyday work process (CI) is an ongoing effort.

Efforts include:

- Improve test coverage for library APIs
- Add and maintain integration tests (with a focus on browser testing)
- Integrate test runs / coverage reports into CI process
- Benchmark libraries, performance improvements for both library execution and tests

1.4.5 W5 - Community Work

There is a high level of engagement from the community with the different `EthereumJS` libraries and there are countless examples for both evolutionary updates as well as high-quality and broadly scoped feature contributions from the community.

We are determined to put substantial ressources here to further support exchange with and engagement from the community.

Related tasks are:

- Help onboard new contributors, give introductory guidance
- Review of Pull Requests
- Accompany community development work
- Management and structuring of issues and PRs
- Responsiveness on communication channels

1.4.6 W6 - Accessibility

Very much related to the community efforts (W5) is the goal of making libraries generally as easily approachable as possible and so to lower the barrier to engage and minimize the need to to do one-to-one explanations on how things work.

Tasks include:

- Provide up-to-date and consistent `API` documentation
- Instructions on environment setup and installation, developer docs
- Easy to recreate and up-to-date examples in `README`
- Common standards and standard documentation (these docs :-)) whenever possible
- Easy to understand, modular and documented source code

CHAPTER 2

Contributing

Everyone is invited to contribute to the `EthereumJS` libraries (see also our *Code of Conduct*). These are some guidelines to help you get started!

2.1 Where to Contribute

2.1.1 Picking up some Issues

There are labelled issues on all our libraries, see e.g. the issue pages of the VM or the Merkle Tree libraries, sorting issues on things like effort needed, priority or type.

Feel free to pick any issue you think is suitable for you to work on, then you might also want to drop a note on the issue page that you are working on the issue.

Some issues are also labelled with `help wanted` and/or `good first issue`, indicating that they are in particular suitable to get started.

2.1.2 Some generic Tasks

There are also various generic tasks which constantly needs help, you can also have a look at the *Ongoing Work Tasks* section to get an overview here.

Many of these things are not listed as issues, but are nevertheless a good place to start especially for new contributors. This includes:

- Improving on the documentation (see: *Documentation*)
- Writing additional tests (see: *Testing*)
- Updating library dependencies (see: *Dependency Management*)

All these things are a good way to gently get in touch with the inner workings of a library without directly have to manipulate production code directly.

2.1.3 Gitcoin

TODO

2.2 How to Start

2.2.1 Introductory Information

Once you have chosen what you want to work on you can actually grab your coffee, take your laptop to a quiet place to work and start hacking!

Have a look at the *Git Guidelines* and the *Workflow Best Practices* sections for some Git and overall work instructions being common practice within the EthereumJS ecosystem.

The *Technical Reference* chapter generally contains some overview information on the development environment, programming language and tools used throughout the libraries. Have a broader look on what is relevant for you to successfully work on your selected task.

2.2.2 Get in Touch

Generally: just get in touch. Early on - see *Team and Contact* section. Feel free to ask everything you need to know, there is no question which shouldn't been asked and there will likely be someone who can give you some guidance along the way.

CHAPTER 3

Technical Reference

This guide gives an overview on common practices and technical standards shared within the `EthereumJS` ecosystem.

3.1 Source Code

3.1.1 Development Runtime

Node.js Version

Runtime environment for development is node.js.

Development should always be possible running the last two `LTS` Node.js versions, see Node.js release schedule table.

Node.js 8	Supported
Node.js 10	Supported

Node.js Features

`TODO`

Notes on `Buffer`, `safe-buffer`, stuff like that.

Node.js Best Practices

Currently agreed-upon best practices on Node.js development:

- Do not use lock files (`package-lock.json`) on repositories (instead add to `.gitignore`), see this discussion

3.1.2 Programming Language

We use both JavaScript and TypeScript in our libraries, with a transition of libraries to `TypeScript` on the way (see: *R18-1 Transition to TypeScript*).

All libraries are transpiled to a lower common denominator JavaScript version (see section below), this section describes what language features are safe to be used in the non-transpiled source code of the libraries.

Main aspects to consider when choosing language version and features for the development code base are:

- `Node.js` compatibility (see also: node.green Website)
- Support by transpilation tools in-use

Note: Not all libraries have transpilation included, have a closer look along when using new language features!

JavaScript

Features supported, encouraged and discouraged from the different `JavaScript` versions:

- ES5
  - All features supported
- ES6 / ES2015
  - All features supported
  - Usage of `let`, `const` encouraged
  - Transition to ES6 `classes` encouraged
- ES2016 (ES7)
  - `TODO`
- ES2017 (ES8)
  - `TODO`

Note: This table does not aim to be complete but just wants to hint to the practically most common pitfall cases!

TypeScript

`TODO`

3.1.3 Linting and Formatting

JavaScript

All `EthereumJS` `JavaScript` libraries use standard.js for code formatting and linting with no extra configuration files added or rules adopted, see the VM repository as an example.

The `standard` dependency in the `devDependencies` section of `package.json` should be upgraded on a regular basis. Count in some time for this, since this normally goes along with some code changes necessary through the introduction of new rules.

Linting can be triggered on the different libraries with an `npm run lint` command being added to `package.json`.

Note: For convenience a `lint:fix` command should be added to the various library `package.json` files.

TypeScript

`TypeScript` libraries are using TSLint for linting and Prettier for code formatting. See the RLP library for a first example (changes might still be located in TypeScript transition PR #37).

Note: It is intended to integrate both linting and formatting config into a shared ethereumjs-config library (see: *ethereumjs-config*), this effort is still ongoing.

3.2 Distribution

3.2.1 Transpilation

Current transpilation target: `ES5`-compatible `JavaScript` code

JavaScript

For `JavaScript` libraries, Babel is used for transpilation, probably the most up-to-date example can be found in the merkle-patricia-tree library.

Note: `TODO`: This section has to be expanded.

TypeScript

For `TypeScript` libraries, transpilation is done through the `TypeScript` compiler `tsc` command line tool.

Note: `TODO`: This section has to be expanded.

3.2.2 Browser Compatibility

`TODO`

3.2.3 Releases

Releases on libraries follow Semantic Versioning, normally releases are published on npm and as a tagged release on GitHub in the `Releases` section.

Every library contains a `CHANGELOG.md` file in the root directory, listing the changes on the respective release versions (see e.g. CHANGELOG.md of the `ethereumjs-util` library), the changelog entry is copied to the GitHub release section on publication of a new release.

Releases go through a PR (see *example PR <https://github.com/ethereumjs/ethereumjs-util/pull/155/files>* on `ethereumjs-util v6.0.0` release), containing the `package.json` version number update, a new CHANGELOG entry and eventually some update on the docs.

3.3 Git Workflow

3.3.1 Branching Model

We are using a feature-centric branching model, the GitHub flow model is coming very much close.

Development of new features is taking place on a dedicated branch and should have some descriptive name for the work done (e.g. `api-doc-fixes`, `remove-vm-accesses-to-statemanager-trie-cache`, `new-bloom-filter-tests`).

Once work on the feature branch is completed and all tests and checks from CI (see *Continuous Integration (CI)*) pass it goes through a review and eventually discussion process and is afterwards merged into a protected `master` branch. The `master` branch should always be stable and theoretically ready for deployment.

3.3.2 Git Guidelines

Some guidelines for the `EthereumJS` libraries when working with `Git` version control:

Feature branch for all PRs

Always do your work on a separate feature branch (see *Branching Model*), this also applies when doing work from an own fork of a library.

This makes it easier for reviewers and others interested to test your code locally by fetching your code changes from your remote feature branch.

Separate PRs for separate Features

If you have separate things you want to change on a library, do separate PRs for this. So if you e.g. have some ideas for how to improve the build process and want to fix some bug from an issue, theses are two separate PRs.

This is a precondition for a successful review of a PR, since a reviewer has a smaller subset of changes and can connect changes undoubtfully to a certain feature. It also avoids the situation where unexpected discussions and disagreements on a certain subfeature set blocks the whole PR with all other changes.

Meaningful Commit History

Make sure that you end up with a meaningful commit history on your work:

- Choose self-descriptive commit messages
- Avoid inconsistent state between commits
- If you do changes correcting your prior committed work, rebase and squash commits afterwards

Note: Rebasing can be a hairy process, if you do for the first time it is highly recommended to do a local backup of your repository.

Note: Rebase work like the above can normally be done with `git rebase -i master` from the feature branch with an up-to-date `master` branch.

Regular Master Rebase

PRs are only reviewed if the branch is up-to-date on the latest `master` changes. Rebase your branch often (with `git rebase master`) and force-push the changes, to make sure that your changes work well on top of the latest commits and tests keep passing.

3.3.3 Workflow Best Practices

Some best practices which turned out to be practical over time and should be followed when working on a new feature:

In doubt: Issue before PR

If you are planning on introducing major feature changes on a library file an issue and describe what you are up to before directly work on a PR. This gives others the chance to discuss around your intended changes and avoids potential further conflicts along the road.

This especially applies for stuff like:

- Introducing new language features (`Promises`,...)
- Changing the API of a library
- Planning security-sensitive changes
- Switch or introduce new tooling

Describe your Work

Take some time to make both the scope of your work and your work process transparent for others. This will ease both discussions and the review process around the work being done.

In particular:

- Do a proper and complete task description on your issue or PR
- Give some regular updates on the current status of your work
- Especially: drop a note once you are ready

3.3.4 Pull Request Reviews

All PRs making changes to the production code base are going through a review process. This will normally take some time and will come along with some back-and-forth between contributor and reviewer until everyone is happy.

3.4 Code Quality

3.4.1 Testing

Test Framework

Most `EthereumJS` libaries use tape for running tests. Have a look at one of the libraries (e.g. merkle-patricia-tree) for reference.

Note: It should be examined if this is a good choice and eventually Mocha should be preferred, see e.g. this comparison.

Code Coverage

For coverage runs nyc is used. Results are passed on to the coveralls.io service for coverage reports on CI runs.

Note: If you stumble over libraries still using `istanbul` as a coverage runner, do an update to `nyc`!

3.4.2 Documentation

On many libraries documentation.js is used for generating an API documentation from JSDoc comments.

Beyond the following documentation should be kept up-to-date:

- `README` with setup and installation instructions
- Usage instructions, up-to-date code examples

3.4.3 Continuous Integration (CI)

Most `EthereumJS` libraries use *Travis CI <https://travis-ci.org/>* for CI runs on every PR submitted. Have a look at a `.travis.yml` file in the repository you are interested in to get an overview on what is run during the CI process.

One exception is the EthereumJS VM which is using `CircleCI` as a platform for performance reasons.

3.5 Security

Security aspects around the EthereumJS libraries should be taken seriously, since many of the libraries are used in production in security-sensitive environments.

3.5.1 Dependency Management

Dependencies are a main source for also importing security vulnerabilities on a library, so the set of dependencies on the libraries should be actively managed and regularly reviewed.

Some guidelines:

Minimal Dependencies

Every introduction of a new dependency on a library should be carefully considered and there has to be solid argument why a new dependency is necessary. This primarily applies for production but also for development dependencies. Dependencies listed in `package.json` should be reviewed on a regular basis if they are still necessary or could be removed.

Established and maintained Dependencies

Only (somewhat) established and actively maintained dependencies should be used on the libraries. Some indicators for a not-so-established dependency:

- Low number of `GitHub` stars or a similar metric
- No commit activity for a longer period of time
- Low download rate on `npm`

Regular Dependency Updates

Dependency versions should be updated on a regular basis, this is also very welcome to be done as a `first-time-contributor` PR. Don't underestimate this task though, since a dependency update almost always come along with some necessary changes on a library. It is recommended to always only do one dependency at a time, since it becomes easier to attribute if things break at some point.

3.6 Shared Library Ressources

The following libraries set up some shared infrastructure for certain purposes.

3.6.1 ethereumjs-testing

The ethereumjs-testing library is a proxy library for the common *Ethereum Tests <https://github.com/ethereum/tests>_* consensus tests. There are additional methods for easily select a specific subset of the tests.

The common test library is integrated as a submodule and there are tagged releases (no publishing to `npm` due to size constraints) which can be used for running the latest tests in `JavaScript` libraries.

3.6.2 ethereumjs-common

The *ethereumjs-common* library provides access to chain and hardfork specific parameters as well as utilities to easier manage hardfork-specific logic within other `EthereumJS` libraries.

3.6.3 ethereumjs-config

`[IN DEVELOPMENT]`

The ethereumjs-config library aims to reduce redundancy on library configuration by providing a unified set of configuration options (e.g. on linting or code formatting) which can be integrated within other libraries.

CHAPTER 4

Roadmap

4.1 Active Projects

4.1.1 R18-1 Transition to TypeScript

There is currently a transition of EthereumJS libraries from JavaScript to TypeScript in the works. This is a somewhat larger effort since it not only requires significant updates to the source code but also comes with changes to the toolchain (e.g. regarding testing, code analysis (linting) and formatting) on all libraries transitioned. This fact nevertheless is an opportunity to rethink parts of tooling and systematically introduce improved procedures along the way.

Bringing type safety to the EthereumJS libraries should bring large mid-term benefits regarding overall security and robustness of the libraries.

Timeline

- `November 2018`
  - Ad-hoc team, tooling discussion, kick-off at Devcon4
- `December 2018`
  - First reference implementation (RLP library)
  - Toolchain best practices draft
- `February 2019`
  - Three more completed transitions, stable toolchain
- `May 2019`
  - All major transitions completed including `VM`, `merkle-patricia-tree`

4.1.2 R18-2 EthereumJS Client

Although popular clients like Geth and Parity already exist, given the popularity of the JavaScript language we have finally started the development of a dedicated JavaScript Ethereum client with fast- and light-sync support. Development started in June 2018 on https://github.com/ethereumjs/ethereumjs-client and reactions from the community have been extremely positive.

Initially, rather than focus on building a consensus-critical client, we want to focus on the following use cases (in order of importance):

- In-Browser/NodeJS research & development (sharding, libp2p, etc.) mainly supported by a modular and extensible (plugin-based) architecture
- In-Browser education applications
- In-Browser/NodeJS client simulations and visualizations
- In-Browser light client (Metamask without Infura)

Generally the EthereumJS client project has larger similarities with the scope of the Trinity project of the Python team. Since JavaScript (like Python) is an extremely popular and widely used language, this will draw in a whole new class of developers who were not able to experiment with and develop on Ethereum client technologies before.

One side goal being nevertheless important is finally to use the client development as a proxy to "harden" the other EthereumJS libraries against a real production environment and serve as a better foundation for testing for our Virtual Machine implementation.

The client project will also build a solid foundation for continued internal research and development efforts. We've already incorporated support for libp2p as an alternate transport to RLPx/Devp2p that enables the in-browser light client use-case. In the future, we hope to implement a Clique PoA engine and test it on the Goerli testnet, and later build a working Ethereum 2.0 stateless client.

At a later point it is also be desired to have a dedicated website for the client (similar to https://geth.ethereum.org/) to have a more visible entry point and source for information around the client for the community.

Timeline

- `Q3 2018`
  - Proof-of-concept chain sync (fast and light)
  - Libp2p networking and browser support
- `Q4 2018`
  - Achieve > 90% code coverage via unit/integration tests
  - Reliable mainnet chain sync (fast and light) including block validation
  - Implement state downloading
- `Q1 2019`
  - Test setup on hive
  - Participate in Kitsunet (https://github.com/MetaMask/mustekala/blob/master/docs/architecture.md#layer-3-kitsunet-peers)
  - Determine Ethereum 2.0 strategy (ShasperJS collaboration? stateless client?)
- `Q2 2019`
  - Alpha release of client

  - Become a signer on the Goerli testnet

4.2 Considered Projects

Projects currently under consideration or in a draft state.

4.2.1 R19-1 Sharding Tools

The all-dominating topic regarding the evolution of Ethereum for the upcoming years will be the implementation of a sharded network together with the integration of the PoS consensus mechanics introduced with Casper.

While it is not intended by the EthereumJS team to provide a full stack solution for these problems on its own, there will be a minimal role for JavaScript implementations in this area to provide a basis for/support:

- 3rd party developer tools with integrated sharding support, e.g. to simulate a sharded deployment
- Sharding R&D on the web
- Sharding chain data structure components (collations, cross-links,...), e.g. for block explorers and other tools

Due to the ongoing research and late-changing specifications in this field, there is still an ongoing debate in the team about the scope of work to be done here. There is consensus though that there will be a minimal targeted need with various useful expansions on this without going too broad in scope.

Note: For this to be moved to the `active` section this needs a more concrete focus first.

4.2.2 R19-2 AssemblyScript (eWASM)

Currently the `eWASM` team is working on the implementation of an upgraded Ethereum virtual machine (VM), replacing the existing EVM with a WebAssembly (WASM) compatible VM, a testnet supporting this is already up and running.

This will allow to write smart contracts in various classical non-blockchain specific languages. One language specifically targeted for support by the eWASM team is AssemblyScript. This language is a subset of `TypeScript` which is basically `JavaScript` with type additions. `TypeScript` is already supported and will become the default language for `EthereumJS` libraries once *R18-1 Transition to TypeScript* is completed.

While `AssemblyScript` is syntactically compatible with `(e)WASM` it will nevertheless take some signifcant high-level work to make this a trusted Ethereum smart contract language.

Tasks in this regard are:

- Define and spec out some practically usable high-level API
- Create code examples
- Build up some tooling infrastructure
- Create helper libraries
- Think about security best practices
- ...

It would be some natural fit for the `EthereumJS` team to take on the high-level part of the `AssemblyScript` work (in contrast to the low-level task to secure `AssemblyScript` to `eWASM` compatibility) due to the familiarity with the language and the close relationship with the eWASM team.

4.2.3 R19-3 eWASM Kernel VM

In a not-too-distant future the current Ethereum Virtual Machine (EVM) will at least gradually and eventually completely be replaced with an eWASM virtual machine.

For this to be prepared the execution engine/kernel of the `EthereumJS` VM implementation needs to be modularized to allow for a pluggable exchange with a new `eWASM` engine. On top of this work bindings have to be created to allow communication with the execution engine implemented by the `eWASM` team.

4.3 Finished Projects

Move projects here once finished (with some note on the outcome).

4.4 Canceled Projects

Move canceled projects here (with some notes on in-between outcome and cancellation reason).

CHAPTER 5

Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at holger@ethereum.org. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

CHAPTER 6

Indices and tables

- genindex
- modindex
- search