Karlsruhe Institute of Technology

Software Engineering Practice

rootJS

Node.js bindings for ROOT 6

Jonas Schwabe
Maxi Früh
Sachin Rajgopal
Christoph Wolff
Christoph Haas
Theo Beffart

supervised by
Dr. Marek Szuba

Contents

1. Purpose

Project Goal

The goal of this project is to create Node.js®[1] bindings for ROOT[2], thanks to which it will become possible to e.g. integrate ROOT into Node.js-based Web applications.
We aim specifically at ROOT 6 because its Low Level Virtual Machine(LLVM)-based C++ interpreter Cling offers many advantages, e.g. just-in-time compilation, over the one available in older ROOT versions.

1.1. Required criteria

The bindings should:

- work on Linux
- allow the user to interact with any ROOT class from the Node.js JavaScript interpreter
- accept C++ code for just-in-time compilation
- update dynamically following changes to C++ internals
- provide asynchronous wrappers for common I/O operations (i.e. file and tree access)

1.2. Optional criteria

The bindings should:

- support the streaming of data in JavaScript Object Notation (JSON) format compatible with JavaScript ROOT
- implement a web server based on Node.js to mimic the function of the ROOT HTTP server
- work OS independent (i.e. support Mac OS X, Linux operating systems)

1.3. Limiting criteria

The bindings should not:

- add any extending functionality to the existing ROOT framework
- necessarily support previous ROOT versions
- necessarily support future ROOT versions

[1] https://nodejs.org/
[2] https://root.cern.ch/

2. Product usage

rootJS will be used to create web-applications that can:

- Expose processed data (that might otherwise be hard to access) and then visualize it locally
- Interact with data both stored somewhere accessible for the server or streamed via remote procedure call (RPC)
- Run on any platform that supports a browser

2.1. Audience

Most users of rootJS will be used to working in Linux and with web servers. At the very least, they will be able to install ROOT and also be proficient in programming languages like JavaScript and C++.

- Scientists (e.g. particle physicists)
- Researchers
- Web-developers interested in creating applications based on ROOT

2.2. Operating conditions

rootJS will be used on servers that run ROOT and have access to the required data sources. As ROOT 6 currently runs on Linux and OS X only, usage of the bindings is limited to those platforms.

3. Product environment

Providing ROOT to Node.js

Node.js bindings for ROOT simplify the creation of client-server based ROOT applications. The bindings offer solutions based on state of the art web technology, especially the separation of data processing and visualization.

3.1. Software

3.1.1 ROOT

ROOT is a software framework for data analysis and I/O. It can be used to process and especially visualize large amounts of scientific data, e.g. the petabytes of data recorded by the Large Hadron Collider experiments every year.
Since the framework comes with an interpreter for the C++ programming language it can be used for rapid and efficient prototyping. It also features a persistency mechanism for C++ objects. Therefore ROOT based applications are extensible and as feature rich as the C++ language itself. A detailed introduction to the ROOT framework can be found in the *ROOT primer*[1] on the CERN website.

Interfacing with ROOT is done dynamically, since ROOT shares all the necessary information in its (global) functions during runtime.

3.1.2 Node.js

Node.js is an open source runtime environment. It is used to develop server-side web applications and may act as a stand alone web server. It uses the Google V8 engine to execute JavaScript code.

The rootJS Binding-API will be developed as a so called native Node.js module written in C++. It interfaces directly with the V8 API to provide (non-blocking) encapsulation of ROOT objects as JavaScript equivalents.

3.2. Hardware

The Bindings simply provide data structures for the encapsulation of ROOT objects and functions. Hence the additional hardware requirements of the bindings themselves should be negligible compared to ROOT's.

Calling a ROOT function via the Binding-API inside a Node.js application will take up a minimal amount of additional resources compared to a direct function call inside a native ROOT application. In conclusion, there are no additional hardware requirements for using the bindings on a computer which was able to run native ROOT applications before - this includes almost any modern Desktop PC.

[1] https://root.cern.ch/root/html534/guides/primer/ROOTPrimer.html

4. Product interface and functions

The rootJS bindings will not have a user interface, neither a graphical user interface nor a command line interface. This section will therefore specify the basic API of rootJS.

/I10/	The rootJS module will expose a JavaScript object containing all accessible ROOT variables, functions and classes.
/I20/	Exposed variables may contain scalar values, in which case they will be accessible in their JavaScript counterparts.
/I30/	Exposed variables may be objects, which are recursively converted to JavaScript objects until only scalar values remain.
/I40/	Exposed variables may be enums, in which case the identifier of the currently selected value is returned instead of the corresponding integer.
/I50/	Every exposed method will be accessible via a proxy method, which handles parameter overloading, as JavaScript does not support overloading. If there is no method to handle the passed arguments, an exception will be thrown.
/I55/	A method may be called with an additional callback method that will be called after the original method has been executed.
/I60/	Exposed classes will be accessible via constructors returning the corresponding objects. A constructor will be accessible through a proxy function to support parameter overloading. If there is no method to handle the passed arguments, an exception will be thrown.
/I65/	A constructor can be called with an additional callback method that will be executed after the object has been constructed.
/I70/	The classes are encapsulated in their namespaces from ROOT. Each namespace is an object containing namespaces or class constructors.
/I80/	Exceptions thrown by ROOT will be forwarded to JavaScript and can be treated as normal exceptions by JavaScript.
/I90/	Global variables are accessible via getter and setter methods to ensure their values are kept in sync with the ROOT framework.

5. Product data

The following data will be stored by the rootJS bindings:

/D10/	All ROOT classes and methods with their corresponding signatures are saved as they are dynamically mapped to JavaScript equivalents. They are also needed to support C/C++ specific features such as method overloading.
/D20/	The ROOT environment state is preserved between calls from Node.js. This means that objects and variables are available until the rootJS session has ended or a reset was triggered explicitly.
/D30/	The application context is derived from *TApplication*[1], storing at least the program name and path.
/D40/	A map of *v8::handle*'s[2] identified by the address of ROOT objects will be stored to allow proper handling of cyclic references, i.e. prevention of stepping down in recursive object creation endlessly.

[1] https://root.cern.ch/doc/master/classTApplication.html
[2] http://izs.me/v8-docs/classv8_1_1Handle.html

6. System Models

6.1. Scenarios

Our bindings do not add to the core functionality of ROOT. Therefore we decided to give some examples on how the bindings might be used.

Scenario Name	WebViewer
Abstract	A browser based GUI for realtime representation of root graphs.
Participating actor instances	WebViewer:Node.js; :ROOT; :rootJS
Flow of events	1. rootJS is up and running initialize has already been executed. 2. WebViewer calls the API method to get graphical output of the data ROOT has currently loaded. (a) rootJS processes the request and calls the corresponding ROOT functionality. (b) rootJS receives ROOT output and streams it to WebViewer 3. WebViewer uses the provided data to display the graph on its GUI (a) Node.js invokes ROOT I/O operations. i. ROOT loads data and provides raw visualization data. (b) Node.js serializes data and streams it to WebViewer. 4. WebViewer receives data and renders it in the browser.

Figure 6.1: WebViewer scenario

6.2. Use Cases

Our bindings do not affect core functionality of ROOT. Therefore we decided to give some examples on how the bindings might be used. The user in this case is a nodeJS based application, that uses our bindings to interact with root. As the actual implementation is not yet finalized, our use cases only contain samples of simple interactions between an app and ROOT.

Scenario Name	eventViewer
Abstract	EventViewer providing a visualisation of experimental data, showing signals particles have produced in the detector. The visualised data can come both from acquired files (which in turn can contain either real data at various degrees of processing or the output of Monte-Carlo simulations) and directly from the detector. This kind of visualisation is typically fairly qualitative but it is very useful to perform a quick eyescan of data, or - quite a common scenario - to monitor live whether the detector actually acquires data properly.
Participating actor instances	:
Flow of events	1. data acquisition 2. preprocessing 3. 3D visualisation 4. stream live data from the detector (a) access the data source (which can use dedicated readout hardware, and even if it is on the network it is typically highly access-restricted) (b) process incoming data in a timely manner 5. publish its data e.g. as JSON over RESTful HTTP or WebSockets 6. web browser for visualisation itself and interaction with the user. 7. render it locally e.g. using WebGL.

Figure 6.2: The X scenario for the use case Y

6.3. Object Models

Figure 6.2 illustrates what the rootJS architecture may look like. Client applications relying on this architecture will incorporate with the ROOT framework through a *ROOTPrototype* object. The API's entry point method *ROOTPrototype::init* uses V8 to create the actual interface to the available variables, functions and classes of ROOT.
Functions provided through this interface internally call *methodProxy* to determine the associated ROOT function via the callee's function name. This allows handling of supplied callback functions by passing them over the *args* parameter.
Constructor functions provided through the interface internally call *classProxy* instead of *methodProxy* to generate encapsulating JavaScript objects through the *ProxyObjectFactory*. A *ROOTPrototype* object provides the *globalGetter* and *globalSetter* methods to access ROOT's global variables. Again interfacing with global objects is done through *ProxyObject*s generated by the *ProxyObjectFactory*.

The *ProxyObjectFactory* instantiates a class realizing the *ProxyObject* interface. The actual class type is given through the supplied *type* parameter. If the *getV8Handle* method is called on *ProxyObject*s encapsulating C++ scalar types (like int, long, string, etc.) it will simply return the corresponding *v8::Handle* with the value at a defined address in memory.

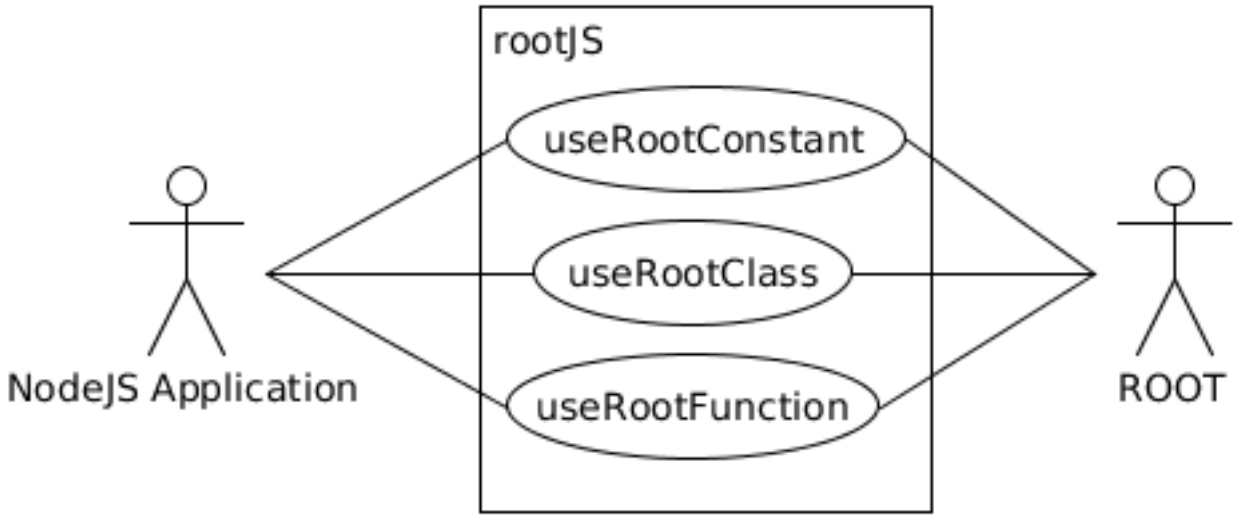

Figure 6.3: use case overview

However, calling the *getV8Handle* method on *ProxyObject*s encapsulating C++ objects will prompt recursive calls to the *ProxyObjectFactory* for every (member) variable the native object holds. This allows to dynamically assemble the encapsulating object. To handle cyclic references a caching mechanism (*ProxyObjectCache*), that stores already created *v8::Handle*s, is used.
A sample process of interfacing with a ROOT object using this architecture is shown in Figure 6.4.

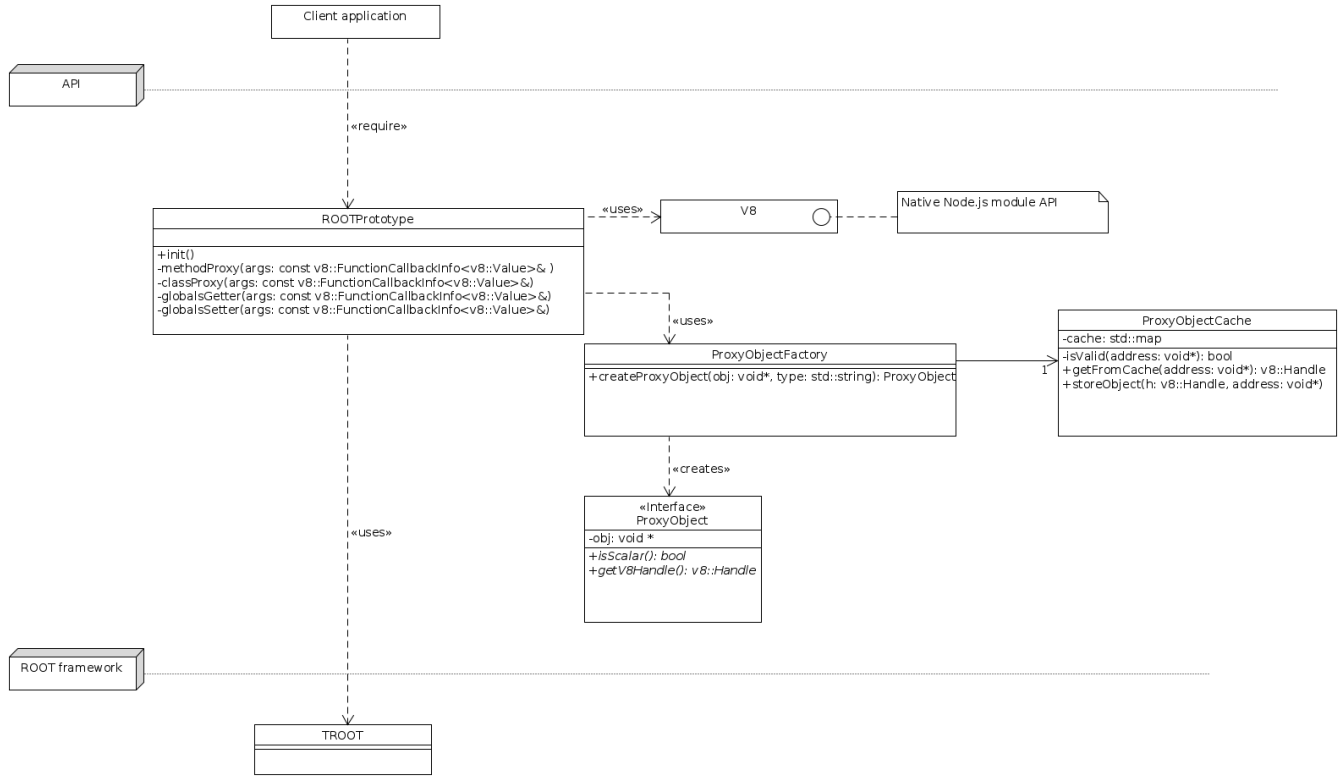

Figure 6.4: basic architecture draft

6.4. Dynamic Models

The following figure shows how rootJS initializes upon being called the first time by a client application. As bindings do not add any functionality of their own, the client application is not further specified. After the bindings are initialized the client application may use any ROOT functionality through a *ROOTPrototype* object provided by the rootJS API.

The following figure shoes how to open a root file via rootJS, after the usual initialization process has been completet (see above) a TFile object is beeing created using the rootJS API. A callback is provided because the file is beeing parsed by ROOT during the opening process, which might take some time depending on file size and complexity. The rootJS bindings call the TFile constructor before returning to the JavaScript application via callback, the constructor is beeing called by the class proxy. After the construction phase rootJS has a valid TFile object, which is then handed over to the CreateProxyObject factory method which creates the correct proxy object. The factory object recursively creates a v8 handle which is returned to the JavaScript code running in node, in this case via callback.

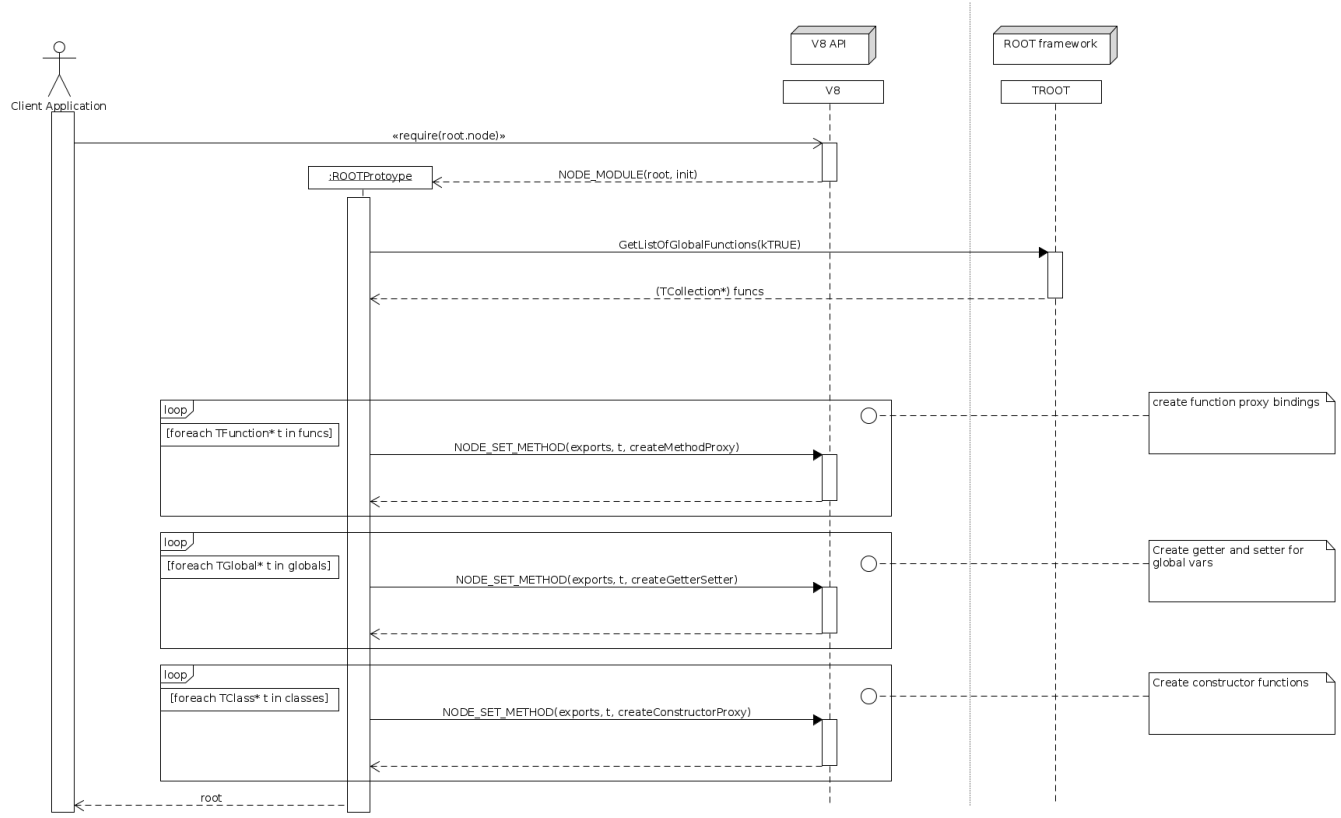

Figure 6.5: basic startup sequence

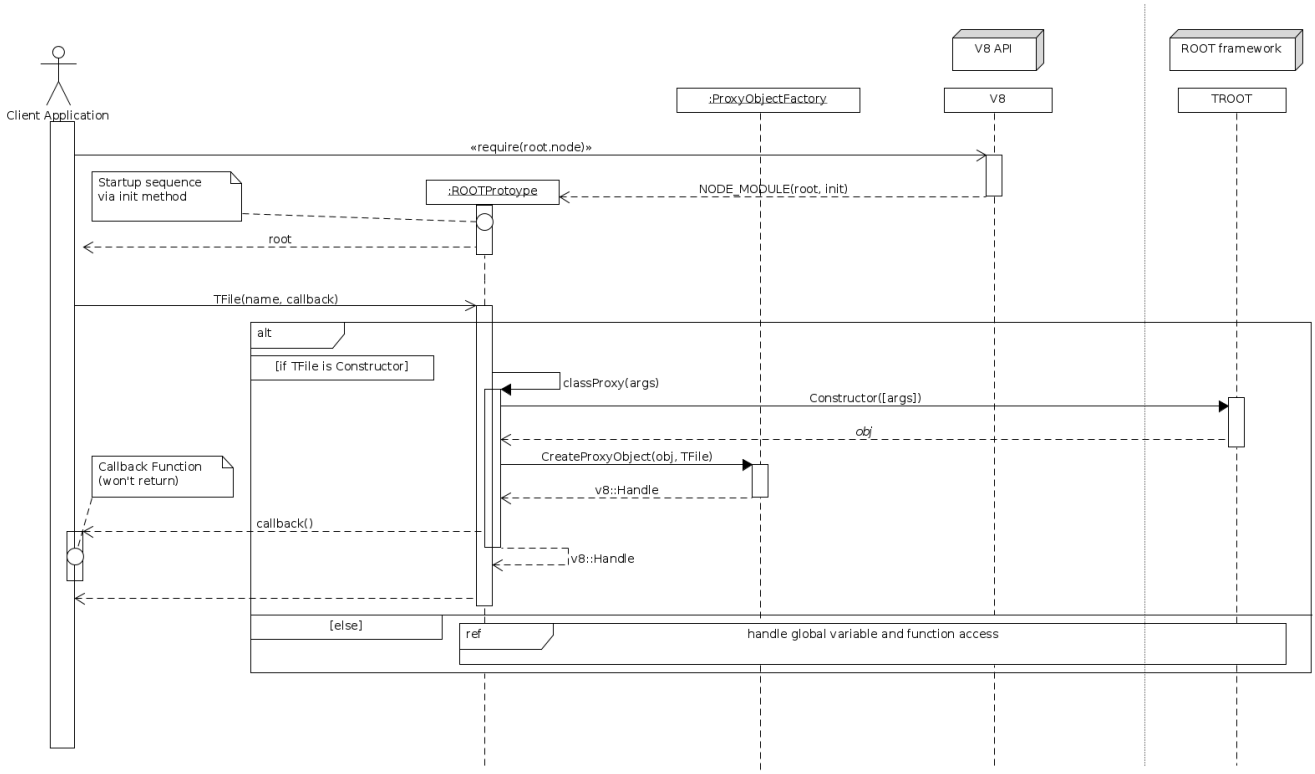

Figure 6.6: basic sequence used for opening a file

7. Global Test Cases

During the development process continuous integration tools will be used to run at least the following test cases:

/T10/	Read all global variables.
/T20/	Write to all global variables that are not constant.
/T30/	Write to all global variables that are constant and ensure the correct Exception is thrown.
/T40/	Create instances of all classes with a public constructor.
/T50/	Call all methods of these objects with valid parameters, where valid means that the data type is correct, a method throwing an exception due to invalid input shall be considered as a passed test, a crash due to e.g. invalid memory read shall be considered as a failed test.
/T60/	Read all public member variables of these classes.
/T70/	Write to all public member variables of these classes that are not constant.
/T80/	Write to all public member variables of these classes that are constant and ensure the correct exception is thrown.
/T90/	Create instances of classes with private constructors and ensure the correct exception is thrown.
/T100/	Apply the test cases described in /T60/ to /T90/ to static members and methods. Ensure correct exceptions are thrown.

8. Quality assurance

We will ensure correct functionality by writing unit tests for every class. Test cases will be run after every git push using continuous integration tools.

Furthermore a function test will be executed, testing if all exposed elements are really accessible and working (see global test cases).

We will use the *Github issue tracker*[1] to track all issues we encounter, even issues that the reporter fixes himself should be tracked. Closing an issue in the issue tracker is only allowed if a test case is provided that both fails before and succeeds after the fix. This ensures that the same bug is not introduced multiple times by different people.

We will check the test coverage after every push to the Github repository. If the coverage decreases on a special method its developer needs to check if there is a branch that is not covered yet and possibly add new test cases for it.

[1] https://github.com/rootjs/rootjs/issues

9. Appendix

9.1. Glossary

Application programming interface (API)

A collection of tools, libraries and documentation for creating software. APIs expose functionality through specified interfaces, which allows to develop independent from the specific implementation

Asynchronous I/O

Using asynchronous I/O, an application does not have to wait for the data transmission to finish, but can continue in execution. This prevents leaving the CPU idle, as I/O operations are usually significantly slower than other tasks.

C++

A general purpose, object oriented programming language and one of the most widely used programming languages.

Compiler

A computer program that translates source code into another language.

Cling

An interactive C++ interpreter with a command line prompt that offers just-in-time compilation.

CERN

The European Organization for Nuclear Research.

enum

A data structure (enumeration) that contains named elements that can be compared to each other. Each element maps to a constant value.

Exception

Exception handling is the process of responding to the occurrence, during computation, of exceptions anomalous or exceptional conditions requiring special processing - often changing the normal flow of program execution.[1]

Framework

Extendable, generic software to be extended by user-written modules.

Git

A Version Control System (VCS).

[1] https://en.wikipedia.org/wiki/Exception_handling

Github

A host for Git repositories.

Google V8 JavaScript

An open-source JavaScript interpreter.

HTTP

The HyperText Transfer Protocol, used for data communication on the Internet.

Interpreter

A computer program that directly executes written instructions, without compiling them first.

JavaScript

An interpreted programming or script language created by Netscape.

JavaScript Object Notation (JSON)

A data format used for exchanging (data) objects between applications. The JSON format is easy to read and write for humans.

Just-in-time (JIT) compilation

Compilation method that tries to speed up interpreter based program execution by dynamically compiling the executed program (parts) at run time.

Language bindings

Expose an API to another programming language. This enables usage of low-level functionality in higher-level environments (such as the ROOT API in Node.js using the rootJS bindings).

Large Hadron Collider (LHC)

World's largest particle collider located in CERN.

Linux

A free and open source computer operating system.

Low Level Virtual Machine (LLVM)

A compiler infrastructure written in C++.

Mac

A Unix based operating system created by Apple.

Method overloading

Declaring a method multiple times within the same namespace using different parameters.

Method signature

A unique declaration which defines the name and the parameter list of a method.

Node.js

A runtime environment for developing server-side web applications written in JavaScript.

Object

An instance of a class containing data, such as variables, functions and data structures.

Operating System

A piece of software managing software and hardware resources, input/output, and also controls the overall operation of the computer system.

ROOT

A framework for data processing; developed at CERN and particularly used by particle physicists.

Remote procedure call (RPC)

Execute a routine in another address space without coding details for the remote interaction.

Stream

Potentially unlimited sequence of data elements made available over time.[2]

Version Control System (VCS)

Version Control Systems are used to track and (if necessary) revert changes made on a software project.

Windows

A DOS based operating system created by Microsoft.

[2] https://en.wikipedia.org/wiki/Stream_(computing)

9.2. Links

The Github repository for this document may be found at https://github.com/rootjs/specifications. The repository for the rootJS project itself is located at https://github.com/rootjs/rootjs.