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-based Web applications.
We aim specifically at ROOT 6 because its Low Level Virtual Machine(LLVM)-based C++ interpreter Cling offers many advantages 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 competent in Linux and web servers. They will be able to install ROOT, and also 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 server-client 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 may be used to process and especially visualize big 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, for rapid and efficient prototyping and a persistency mechanism for C++ objects, ROOT based applications are extensible and as feature rich as the C++ language itself. A detailed introduction to the ROOT framework may be found in the *ROOT primer*[1] on the CERN website.

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

3.1.2 Node.js

Node.js is an open source runtime environment. Node.js 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 the JavaScript code.

The Binding Application programming interface (API) to be developed will be 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

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

Basically calling a ROOT function via the Binding-API inside a Node.js application really should not take up a huge 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 that 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 API of rootJS.

/I10/	The module will expose a JavaScript object containing all accessible root variables, functions and classes
/I20/	Exposed variables might contain scalar values, in this case they will be accessible in their JavaScript counterparts
/I30/	Exposed variables might be objects, which are recursively converted to JavaScript objects until there are only scalar values
/I40/	Exposed variables might be enums. In this 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 can be called with an additional callback method that will be called after the original method has been executed
/I60/	Exposed classes will be accessible as a constructor returning the object. The constructor will be proxied 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 called 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 by JavaScript as normal exceptions
/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

/D10/	All ROOT classes and methods with corresponding signatures 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 explicitly triggered.
/D30/	Application context derived from TApplication, storing at least the program name and path
/D40/	A map of V8 handles identified by the address of root objects, without this we could not handle cyclic references. We would step down in the recursive creation endlessly if we do not use a cache.

6. System Models

6.1 Scenarios

The scenarios are based on Node.js based web applications on a server running root and rootJS Bindings do not extend the basic functionality of the underlying framework.
We therefore only assume a fictional application, using our API, to be the user of rootJS.

Scenario 1: //TODO fix layout and create UML WebViewer, a browser based GUI for realtime representation of root graphs rootJS is up and running initialize has already been executed WebViewer calls the API method to get graphical output of the data ROOT has currently loadeed rootJS processes the request and calls the corresponding ROOT functionality rootJS receives ROOT output and streams it to the Webviewer client webViewer uses the provided data to display the graph on its GUI

Node.js invokes ROOT I/O operations

ROOT loads data and provides raw visualization data

Node.js serializes data and streams it to the web viewer

Web viewer receives data and renders it in the browser

6.2 Use Cases

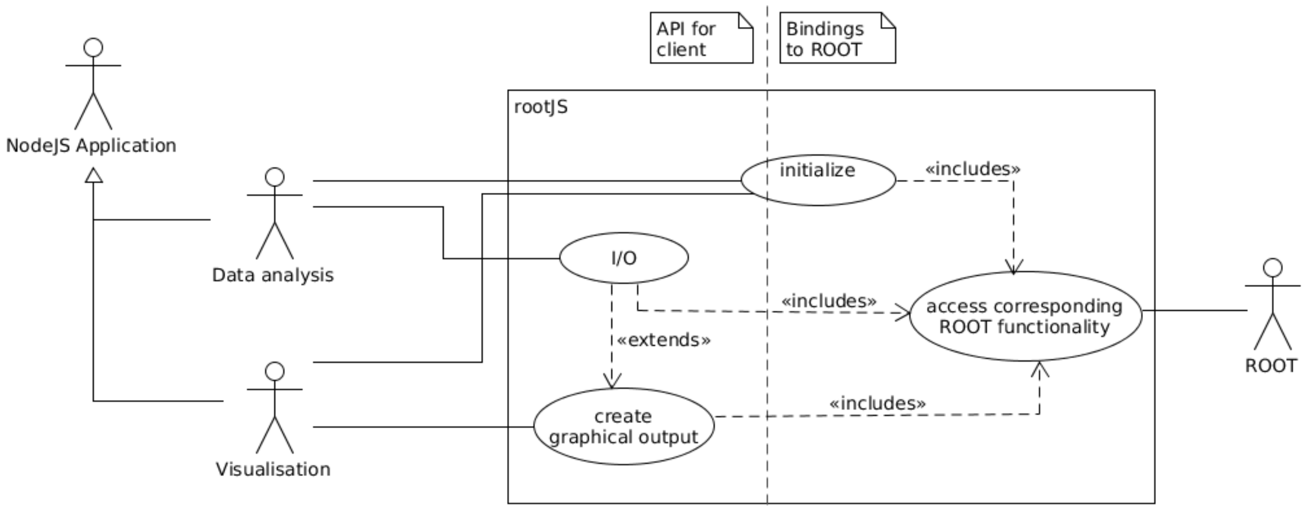

Figure 6.1: Usecase samples

6.3 Object Models

Figure 6.2 illustrates what the bindings architecture may look like. *RootApplication::init* exposes the available variables, functions and classes via V8 to JavaScript. Exposed functions call *methodProxy*, which uses the callee function name to determine the ROOT function that should be called. This will also handle callbacks which are passed via the *args* parameter. The *classProxy* is called instead of *methodProxy* when the specified function is a constructor. This will use *ProxyObjectFactory* to generate a JavaScript object. The *globalGetter* and *globalSetter* method are called when a global variable should be accessed. Again the *ProxyObjectFactory* will be used in this case to generate corresponding JavaScript objects.

The *ProxyObjectFactory* creates an instance of a subclass of *ProxyObject*, the acutal subclass will be decided by using the *type* parameter. If the *ProxyObject* contains a scalar type (like int, long, string, ...) the *getV8Handle* method will simply return the corresponding *v8::handle* with the value at the defined address. If the *ProxyObject* contains an object, *getV8Handle* will recursively call the *ProxyObjectFactory* for every element it contains and builds a JavaScript object. To handle reference circles we use a caching mechanism that stores already created *v8::handle*s to reuse them.

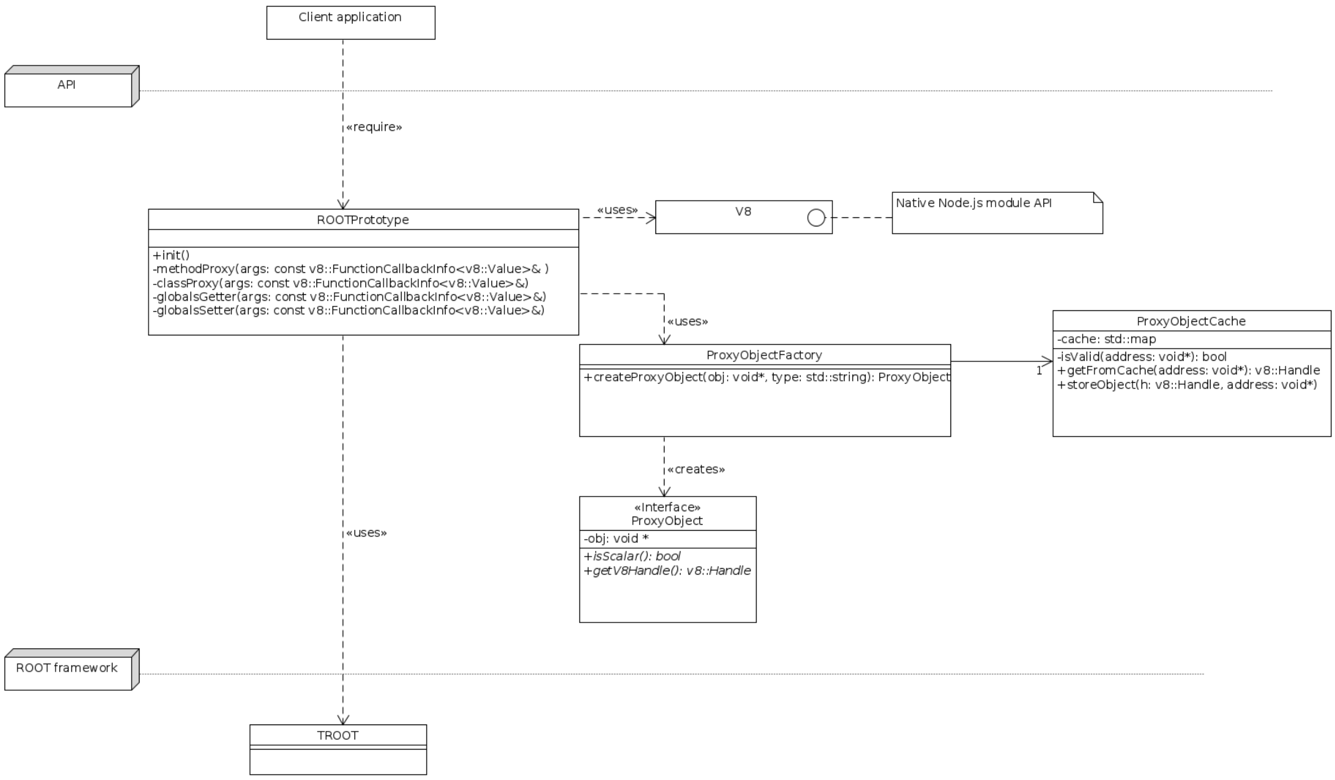

Figure 6.2: basic architecture draft

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 the roootJS API.

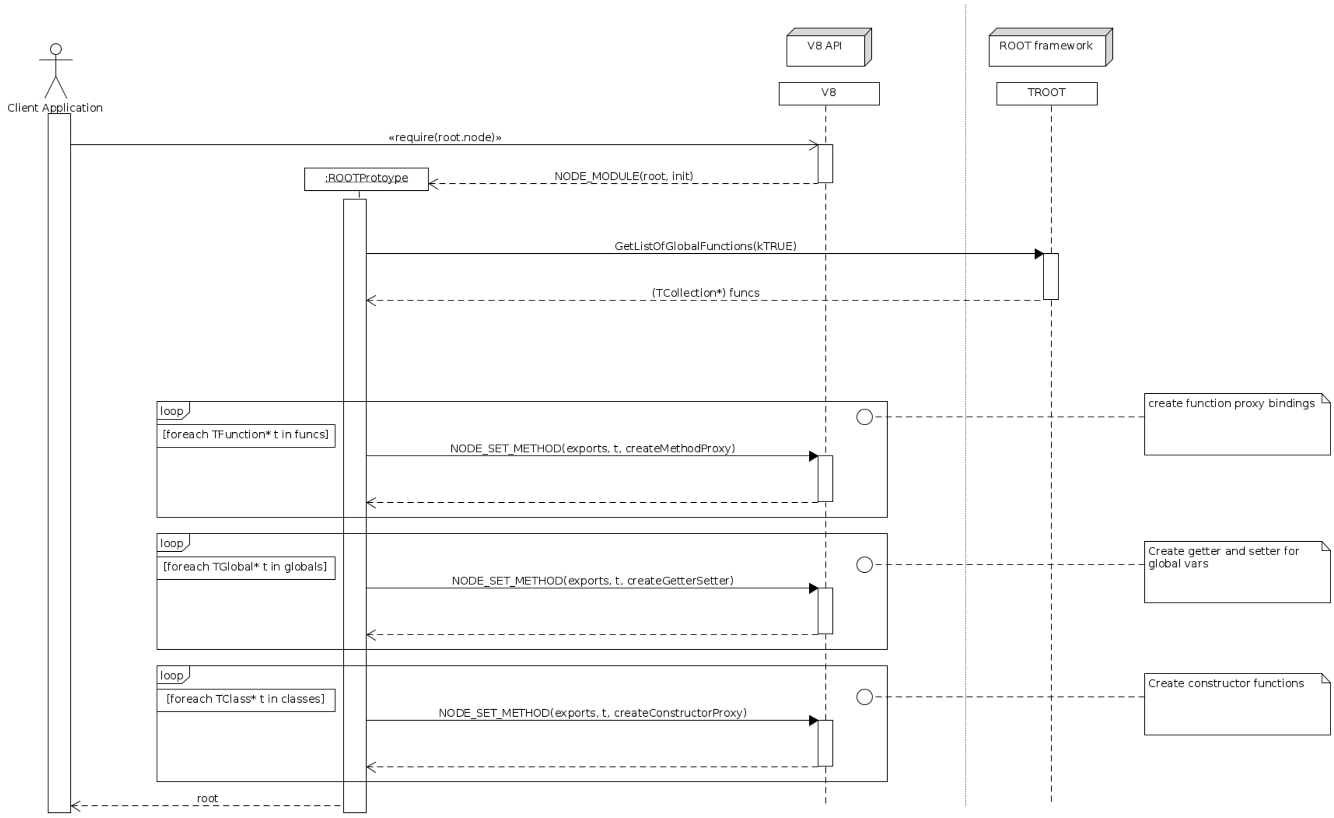

Figure 6.3: startup sequence

7. Global Test Cases

During the development process we will use continuous integration tools, running 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 a passed test, a crash due to e.g. invalid memory read shall be considered a fail
/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/	The same for static members and methods

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 the developer of this method needs to check if there is a branch that is not covered yet and add a new test case for this branch.

[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 and uses a just-in-time compiler.

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

CERN The European Organization for Nuclear Research

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.

Github A host for Git repositories.

Git A VCS Version Control System)

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.

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

JavaScript Object Notation (JSON) A data format which is easy to read and write for humans.

Just-in-time (JIT) compilation

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 Defines the data a method takes in and puts out.

Node.js A runtime environment for developing server-side web applications written in Javascript.

Object

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

Platform

ROOT A framework for data processing, particularly for particle physicists. ROOT was developed by CERN C++.

RPC Remote procedure call. Execute a routine in another address space without coding details for the remote interaction

Stream

Web server

Windows A DOS based operating system created by Microsoft.

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

9.2 Links

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