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, in simplified terms, just provide data structures for encapsulation of ROOT objects or rather functions, the hardware requirements of the bindings themselves should be negligible.
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 do not have a usual interface, there will neither be a graphical user interface nor a command line interface. This section will therefore specify the application programming interface.

/I10/	The module will expose a JS object containing all accessible root variables, functions and classes
/I20/	Exposed variables might contains scalar values, in this case they will be accessible in their JavaScript counterparts
/I30/	Exposed variables might be objects, these objects 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, an Exception will be thrown if there is no method to handle the passed arguments
/I55/	A method can be called with an additional callback method that will be called after the method has been executed
/I60/	Exposed classes will be accessible as a construction method, returning the object, the construction method will be proxied in order to support parameter overloading, an exception will be thrown if there is no method to handle the passed arguments
/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 handled the usual way
/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 deriving from TApplication, storing at least the correct program name and path

6. System Models

6.1 Scenarios

Node.js based web application on a server running root and rootJS

Webviewer launches and provides a GUI to its end user
Webviewer requests data for visualization by calling rootJS
    Node.js invokes root i/o operations
        ROOT loads data and provides raw visualiztion data
    Node.js serializes data and streams it to the web viewer
Webviewer receives data and renders it in the browser

6.2 Use Cases

6.3 Object Models

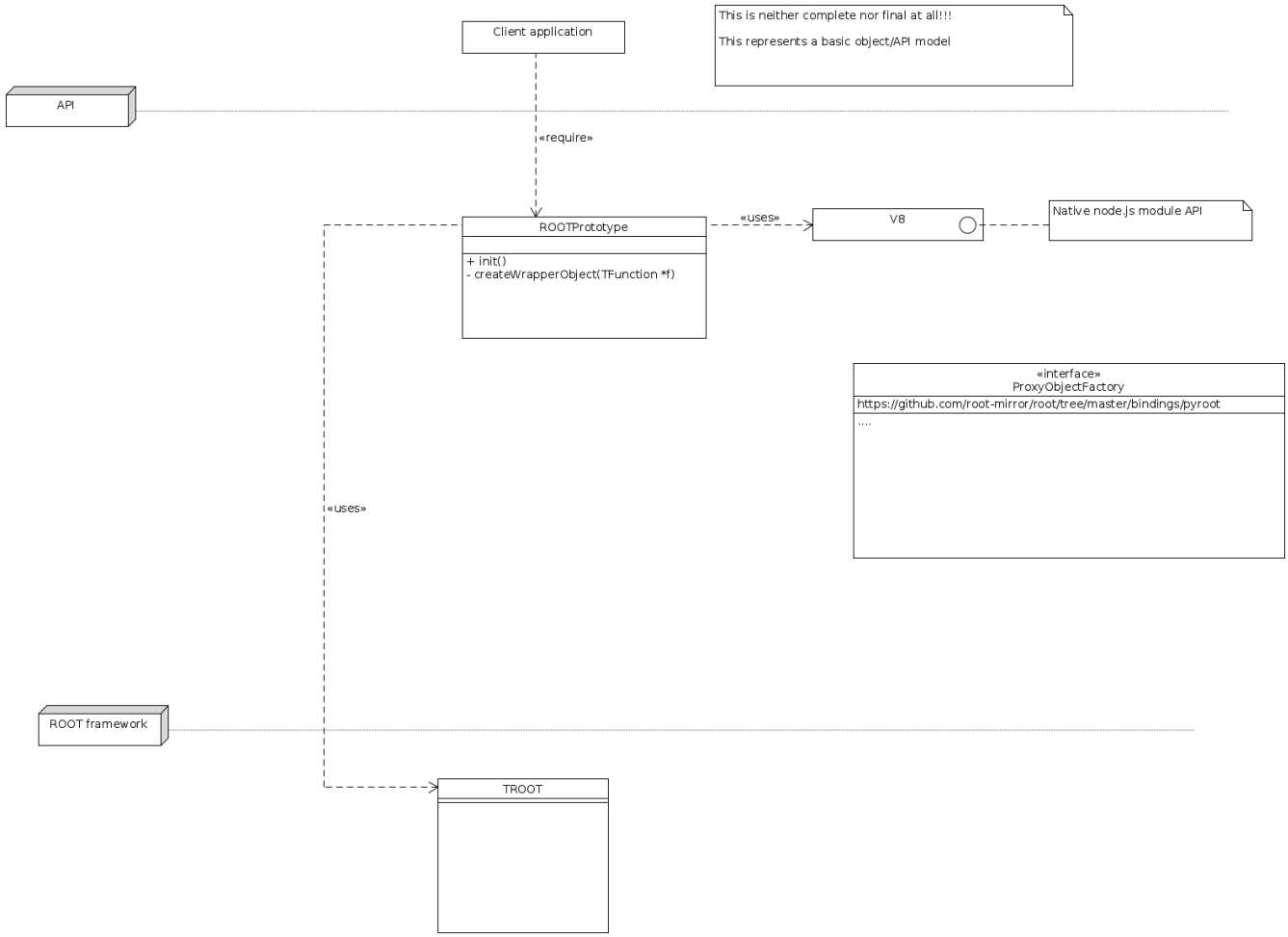

Figure 6.1: basic architecture draft

6.4 Dynamic Models

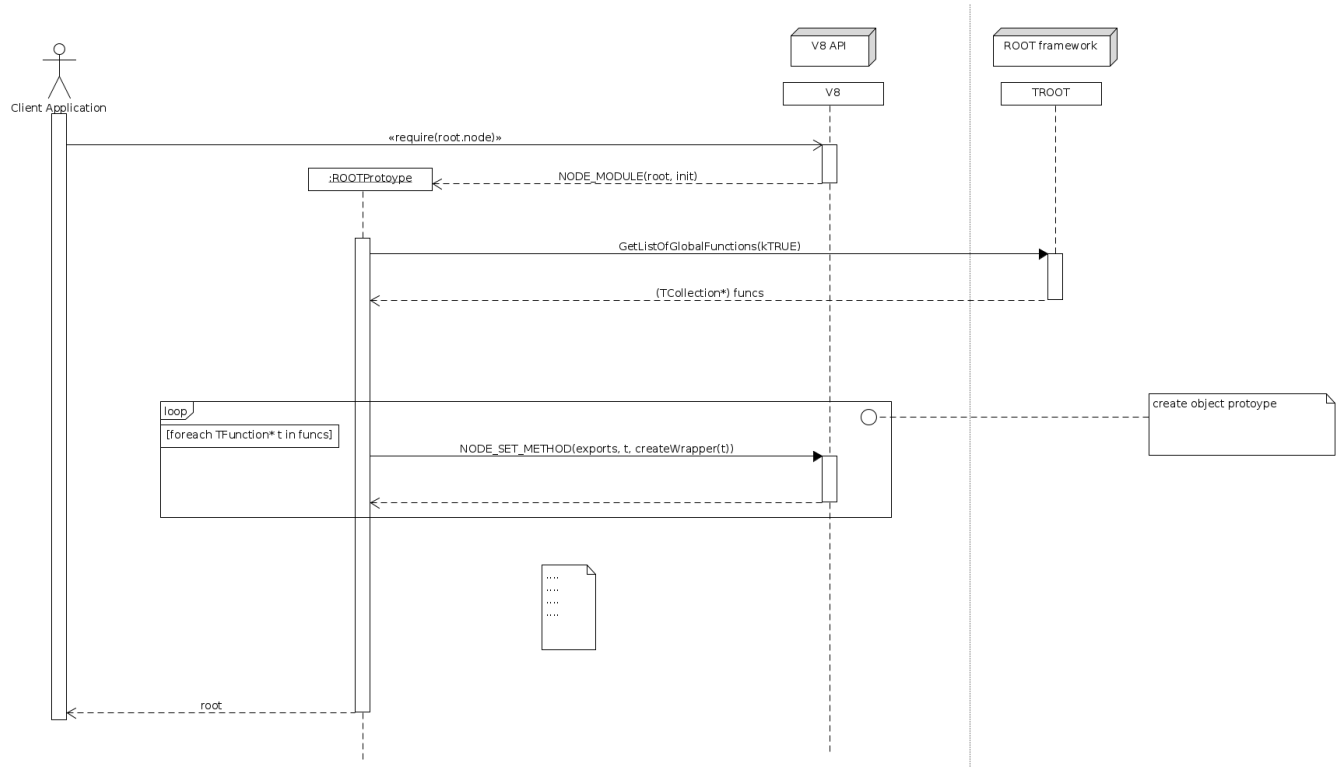

Figure 6.2: startup sequence

7. Global Test Cases

During the development process we will use continuous integration tools, running at least the following testcases:

/T10/	Read all global variables
/T20/	Write to all global variables which are non const
/T30/	Write to all global variables that are const 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 datatype 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 non 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 push using continuous integration tools.

Further 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 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 fails before and works after the fix. We do this to make sure 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, when 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.

9. Appendix

9.1 Glossary

Application programming interface (API)

Asynchronous

Binding API

C++ It is a general purpose, object oriented programming language and is one of the most widely used programming languages.

Cling An interactive C++ interpreter with a command line prompt and uses a just-in-time compiler.

enums

European Organization for Nuclear Research (CERN)

Exception

Expose

Framework

Github

Google V8 JavaScript

HTTP

Input/Output (I/O)

Interpreter

JavaScript An interpreted programming or script language created by Netscape.

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

Just-in-time (JIT) compilation

Large Hadron Collider (LHC)

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

Method signature

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

Stream

Web server

Windows A DOS based operating system creating by Microsoft.