Node.js Bindings for ROOT6

November 22, 2015

Contents

1. Purpose

Project Goal The goal of this project is to create Node.js bindings for ROOT, 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 LLVM-based C++ interpreter Cling offers many advantages over the one available in older ROOT versions.

1.1 Must-have 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 Nice-to-have criteria

The bindings should:

- support the streaming of data in JSON format compatible with JavaScript ROOT
- work OS independent (i.e. support MAC and Windows operating systems)

1.3 Criteria for demarcation

The bindings should not:

- add any extending functionality to the existing ROOT framework.

2. Product usage

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

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

2.1 Audience

1. particle physicists
2. CERN
3. 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.

3. Product environment

3.1 Software

3.2 Hardware

4. Product functions

/BID/	Description

5. Product data

6. Product deliverables

/FID/	Description

7. Product interface

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, insted 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 ran
/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/	What happens when a value changes in root or a global variable is beeing changed in node? How can this be synced? e.g. call gApplication-¿SetName() from node, what happens to the exposed global variable gProgName?, We might add a sync method that does a bidirectional sync, or use getters and setters for global varibales which will be proxied every time, so that the js application does not have to hold any state.

8. Global testcases

/TID/	Description

9. Quality assurance

10. Appendix