Karlsruhe Institute of Technology

Software Engineering Practice

WINTER TERM 2015/2016

rootJS - Implementation Report

Node.js bindings for ROOT 6

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

supervised by
Dr. Marek Szuba

Contents

1. Introduction

1.1. About rootJS

The purpose of creating Node.js bindings of ROOT, called rootJS, is to enable users to integrate ROOT in Node.js programs, such as Node.js based web servers.

2. Changes in the Architecture

2.1. The class diagrams in comparison

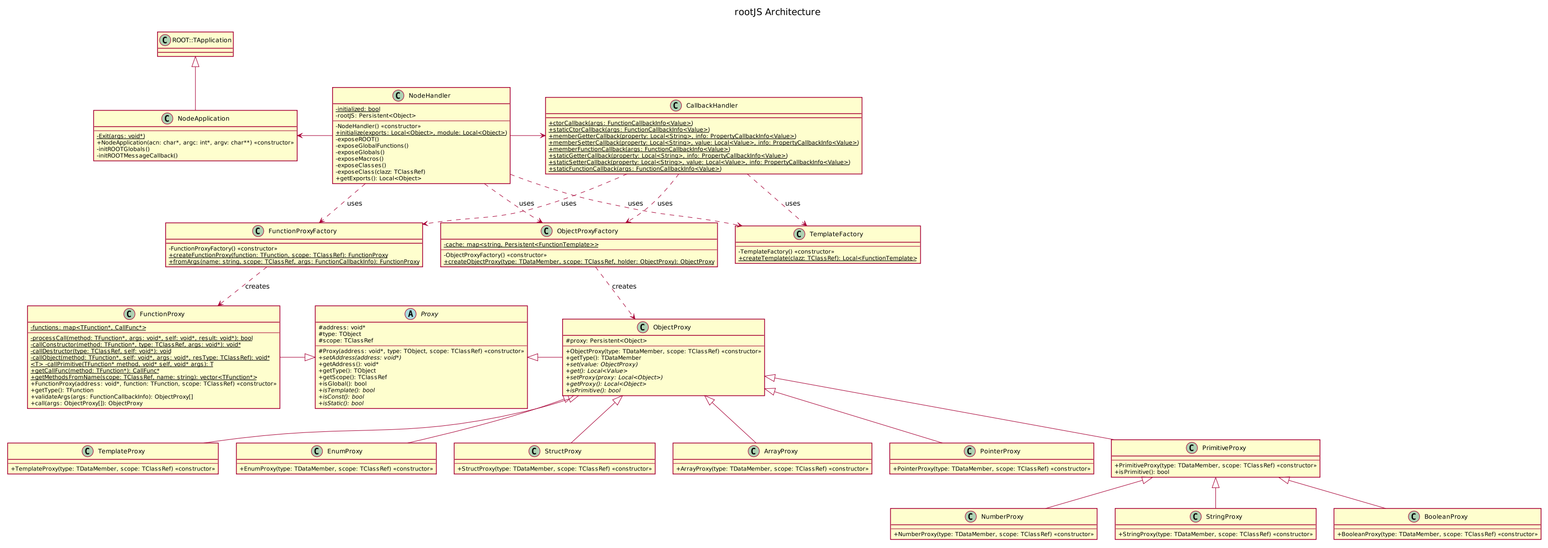
rootJS Architecture
ROOT::TApplication
NodeApplication
-Exit(args: void*)
+NodeApplication(acn: char*, argc: int*, argv: char**) «constructor»
-initROOTGlobals()
-initROOTMessageCallback()
NodeHandler
-initialized: bool
-rootJS: Persistent<Object>
-NodeHandler() «constructor»
+initialize(exports: Local<Object>, module: Local<Object>)
-exposeROOT()
-exposeGlobalFunctions()
-exposeGlobals()
-exposeMacros()
-exposeClasses()
-exposeClass(clazz: TClassRef)
+getExports(): Local<Object>
CallbackHandler
+ctorCallback(args: FunctionCallbackInfo<Value>)
+staticCtorCallback(args: FunctionCallbackInfo<Value>)
+memberGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)
+memberSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)
+memberFunctionCallback(args: FunctionCallbackInfo<Value>)
+staticGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)
+staticSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)
+staticFunctionCallback(args: FunctionCallbackInfo<Value>)
uses
uses
uses
uses
uses
uses
FunctionProxyFactory
-FunctionProxyFactory() «constructor»
+createFunctionProxy(function: TFunction, scope: TClassRef): FunctionProxy
+fromArgs(name: string, scope: TClassRef, args: FunctionCallbackInfo): FunctionProxy
ObjectProxyFactory
-cache: map<string, Persistent<FunctionTemplate>>
-ObjectProxyFactory() «constructor»
+createObjectProxy(type: TDataMember, scope: TClassRef, holder: ObjectProxy): ObjectProxy
TemplateFactory
-TemplateFactory() «constructor»
+createTemplate(clazz: TClassRef): Local<FunctionTemplate>
creates
creates
FunctionProxy
-functions: map<TFunction*, CallFunc*>
-processCall(method: TFunction*, args: void*, self: void*, result: void*): bool
-callConstructor(method: TFunction*, type: TClassRef, args: void*): void*
-callDestructor(type: TClassRef, self: void*): void
-callObject(method: TFunction*, self: void*, args: void*, resType: TClassRef): void*
<T> -callPrimitive(TFunction* method, void* self, void* args): T
+getCallFunc(method: TFunction*): CallFunc*
+getMethodsFromName(scope: TClassRef, name: string): vector<TFunction*>
+FunctionProxy(address: void*, function: TFunction, scope: TClassRef) «constructor»
+getType(): TFunction
+validateArgs(args: FunctionCallbackInfo): ObjectProxy[]
+call(args: ObjectProxy[]): ObjectProxy
Proxy
#address: void*
#type: TObject
#scope: TClassRef
#Proxy(address: void*, type: TObject, scope: TClassRef) «constructor»
+setAddress(address: void*)
+getAddress(): void*
+getType(): TObject
+getScope(): TClassRef
+isGlobal(): bool
+isTemplate(): bool
+isConst(): bool
+isStatic(): bool
ObjectProxy
#proxy: Persistent<Object>
+ObjectProxy(type: TDataMember, scope: TClassRef) «constructor»
+getType(): TDataMember
+set(value: ObjectProxy)
+get(): Local<Value>
+setProxy(proxy: Local<Object>)
+getProxy(): Local<Object>
+isPrimitive(): bool
TemplateProxy
+TemplateProxy(type: TDataMember, scope: TClassRef) «constructor»
EnumProxy
+EnumProxy(type: TDataMember, scope: TClassRef) «constructor»
StructProxy
+StructProxy(type: TDataMember, scope: TClassRef) «constructor»
ArrayProxy
+ArrayProxy(type: TDataMember, scope: TClassRef) «constructor»
PointerProxy
+PointerProxy(type: TDataMember, scope: TClassRef) «constructor»
PrimitiveProxy
+PrimitiveProxy(type: TDataMember, scope: TClassRef) «constructor»
+isPrimitive(): bool
NumberProxy
+NumberProxy(type: TDataMember, scope: TClassRef) «constructor»
StringProxy
+StringProxy(type: TDataMember, scope: TClassRef) «constructor»
BooleanProxy
+BooleanProxy(type: TDataMember, scope: TClassRef) «constructor»

Figure 2.1: The rootJS class diagram at the conclusion of the design phase

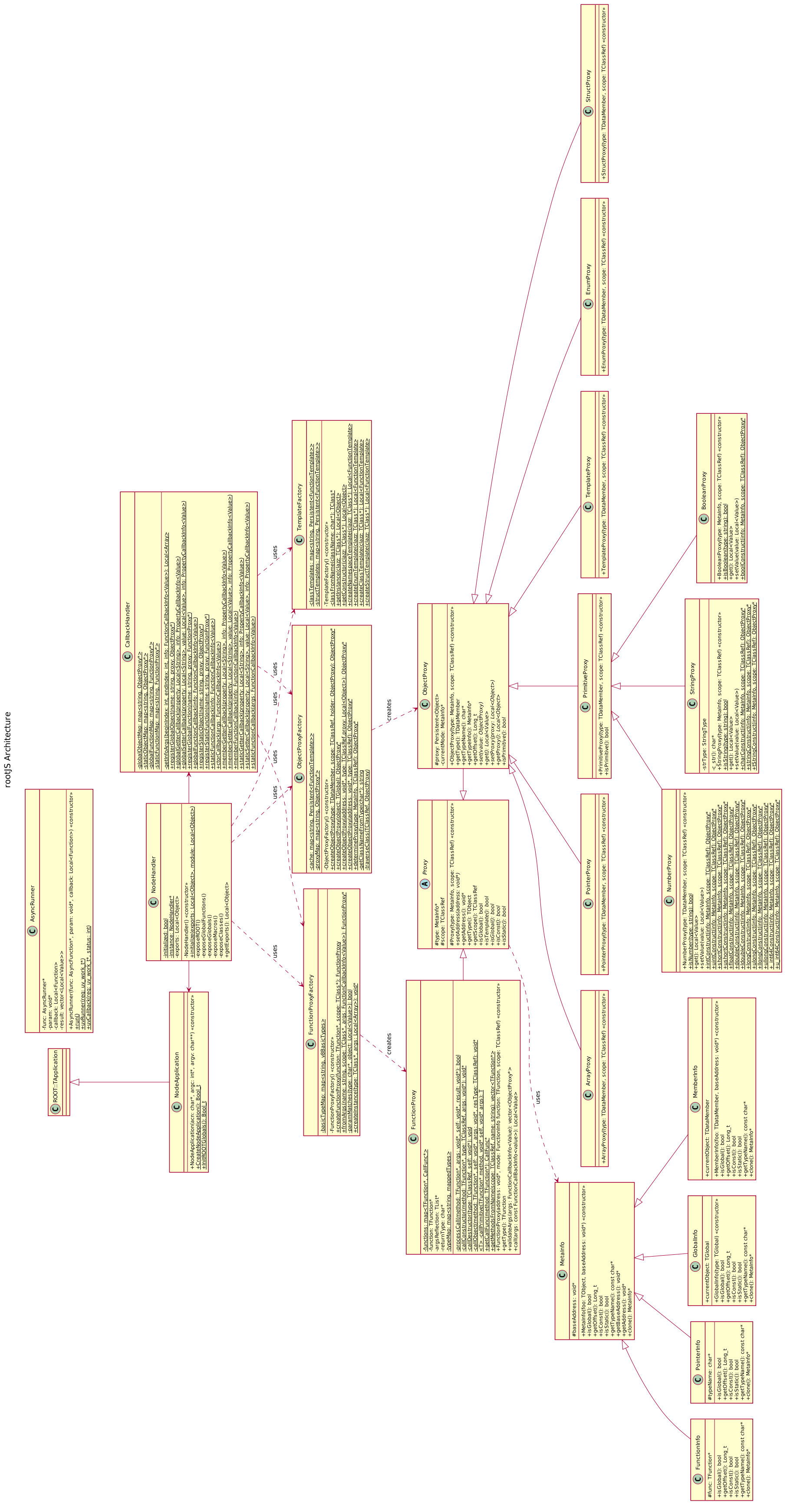

Figure 2.2: The current rootJS class diagram

3. Criteria

3.1. Required Criteria

These are the Required Criteria as stated in the functional specification:

- work on Linux

This is rather straightforward as long as the user has installed the required dependencies.

- allow the user to interact with any ROOT class from the Node.js JavaScript interpreter

The user is able to interact with any ROOT class by requesting any class with rootJS. The following is an extract from the description of the NodeHandler in the module guide, which describes how rootJS is started up:

```
// JavaScript: Load ROOT bindings in JavaScript
var root = require(rootJS.node);

// C++: Expose the initialize method as the main entry point
NODE_MODULE(rootJS, initialize)
```

If the user would then like to request TMath::Pi() from ROOT, then they would have to do the following:

```
var root = require(rootJS.node);
var pi = root.TMath.Pi();
```

The user would then have access on the TMath::Pi() from ROOT.

- accept C++ code for just-in-time compilation

The user will have access to the Node.js interpreter and can enter single lines of code in it. These lines of code will be interpreted by rootJS and rootJS will call

- update dynamically following changes to C++ internals

- provide asynchronous wrappers for common I/O operations (i.e. file and tree access)

While no specifics were written about asynchronous processes in the module guide, two variations were considered. The first variation would utilize ROOT's inbuilt multi-threading environment. ROOT contains a TThread class which is similar to the std::thread class in C++, but is adjusted to ROOT. The second variation would use libuv, a software library that provides asynchronous even notification and was mainly designed for Node.js.

3.2. Optional Criteria

These are the Optional Criteria as stated in the functional specification:

- 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)

3.3. Limiting Criteria

These are the Limiting Criteria as stated in the functional specification:

- add any extending functionality to the existing ROOT framework

No code was written to implement any new features which are not already existing in the ROOT framework.

- necessarily support previous ROOT versions

ROOT 6 uses the LLVM-based C++ interpreter Cling, while ROOT 5.34 uses the C++ interpreter CINT. rootJS was designed with ROOT 6 in mind and many of the function calls, etc. will only work with Cling. As of yet, rootJS has not been tested not been test on older ROOT versions, but it is very unlikely that rootJS will function with ROOT 5.34.

- necessarily support future ROOT versions

If future versions of ROOT also utilize Cling, rootJS might be able to support them. Therefore it is likely that rootJS should support all versions of ROOT 6.

4. Unit Tests