

KARLSRUHE INSTITUTE OF TECHNOLOGY

SOFTWARE ENGINEERING PRACTICE

WINTER TERM 2015/2016

rootJS

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	NodeApplication	2
1.1	initialize	2
1.2	getListOfClasses	3
1.3	exposeClasses	4
1.4	classProxy	5
1.5	getListOfFunctions	6
1.6	exposeFunctions	7
1.7	functionProxy	8
1.8	getListOfVariables	9
1.9	exposeVariables	10
1.10	getterProxy	11
1.11	setterProxy	12
1.12	additional private methods	13
1.13	additional public methods	13
2	ProxyObjectFactory	14
2.1	createProxyObject	14
3	ProxyObject	15
3.1	isScalar	15
3.2	getV8Handle	15
4	ProxyObjectCache	16
4.1	invalidate	16
4.2	put	17
4.3	get	18
5	Appendix	19
5.1	Glossary	19

1. NodeApplication

The *NodeApplication* class is the main entry point of the bindings, during the initialisation phase it exposes all functions, classes and variables to node, further it contains the proxy methods called by node

1.1. initialize

<i>Name</i>	<code>NodeApplication::initialize(Local<Object> exports)</code>
<i>Visibility</i>	Public
<i>Parameters</i>	exports: exports object from the node framework, all methods and variables created in the scope of this object will be available in the node program
<i>Return value</i>	<i>none</i> functions and variables will be attached to the exports object
<i>behavior</i>	Creates a NodeApplication and calls the private expose methods

This is the function that is being called when the module is required in node, it is being referred to, from the node macro

```
NODE_MODULE(rootJS, initialize)
```

Where rootJS is the name of the node module and initialize is a pointer to the function defined in this section.

1.2. getListOfClasses

<i>Name</i>	<code>NodeApplication::getListOfClasses()</code>
<i>Visibility</i>	Private
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	TCollection of TClass A collection containing all classes available in the ROOT framework
<i>behavior</i>	The Cling interpreter is scanned for classes, the result

During the specification phase we decided to use *TRoot::GetListOfClasses*¹ to retrieve a list of all classes that should be available in the node interpreter. The method returns a *TCollection*, containing *TClasses*. *TClass* provides all information needed to generate JavaScript objects by using methods like *GetListOfMethods*.

It turned out that *TRoot::GetListOfClasses* only returns a few classes. To work around this issue PyROOT uses *gInterpreter* in case the class does not exist in the *TCollection*. While instantiating a node module a JavaScript object is provided that will be filled with methods or variables during the module initialisation. Methods that are not bound to the JavaScript object during this phase will not be callable, we are not able to handle class initialisations of classes which are not in the *TRoot::GetListOfClasses* collection because we do not know they are there before a user calls them. Whereas the python bindings fire up the interpreter when an unknown class should be instantiated.

When running the ROOT interactive interpreter *TRint*² all classes can be found using the autocompletion feature. Looking at the source code, it turned out to use *gInterpreter->GetMapfile()->GetTable()* to retrieve a list of all the classes and *gClassTable*³ which is a *HashMap*, to retrieve the corresponding *TClass* objects.

With this approach we have access to all classes that can be accessed by Cling - we would therefore support the same set of features the ROOT interactive interpreter does.

Check if *gClassTable* contains all classes from *gROOT->GetListOfClasses*, can we drop *gROOT->GetListOfClasses* completely?

¹<https://root.cern.ch/doc/master/classTR00T.html#abc882c118c351b40f8b96de4afabe5f7>

²<https://root.cern.ch/doc/master/classTRint.html>

³<https://root.cern.ch/doc/master/classTClassTable.html>

1.3. exposeClasses

<i>Name</i>	<code>NodeApplication::exposeClasses(Local<Object> exports)</code>
<i>Visibility</i>	Private
<i>Parameters</i>	exports : exports object from the node framework, all methods and variables created in the scope of this object will be available in the node program
<i>Return value</i>	<i>none</i>
<i>behavior</i>	After running this method all class constructors are added to the exports object

For every ROOT class we run

```
NODE_SET_METHOD(exports, className, classProxy)
```

exports is the JavaScript object which we can use to pass methods and data to node

className is the name of the class in the current iteration

classProxy is the function that should be called everytime a class is constructed via JavaScript

All constructors are now available using the exports object.

1.4. classProxy

<i>Name</i>	<code>NodeApplication::classProxy(const v8::FunctionCallbackInfo<v8::Value>& args)</code>
<i>Visibility</i>	Public
<i>Parameters</i>	args : arguments passed by node
<i>Return value</i>	<i>none</i> , return value will be passed via the args object
<i>behavior</i>	The passed arguments will be converted to C++ native arguments or root objects, the list of all classes will be checked for a class with the given name. A matching constructor will be selected and called. If the last parameter is a JavaScript function, it will be used as a callback and the call does not block and calls the callback function after finishing. Results will be converted to JavaScript objects or natives.
<i>Exceptions</i>	In case there is no matching constructor, an exception will be thrown. Exceptions from ROOT will be forwarded

We only use one proxy for every class to minimize overhead. This is possible because we can store data in a secret area that is not visible via JavaScript, in there we will store data describing the constructor call. The other option would be to have a proxy class, containing the proxy method and meta information (like the class name or a TClass reference). The proxy method of a concrete object would then be exported to JavaScript.

This would not be faster as we use hashing to find classes, but would consume more memory.

1.5. getListOfFunctions

<i>Name</i>	<code>NodeApplication::getListOfFunctions()</code>
<i>Visibility</i>	Private
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	TCollection of TFunction A collection containing all functions available in the ROOT framework
<i>behavior</i>	<code>gROOT->GetListOfGlobalFunctions</code> will be used to get the TCollection

This method will more or less bypass the results of `gROOT->GetListOfGlobalFunctions` and is only there if we run into a situation where we need to manipulate the collection before exporting the functions.

1.6. exposeFunctions

<i>Name</i>	<code>NodeApplication::exposeFunctions(Local<Object> exports)</code>
<i>Visibility</i>	Private
<i>Parameters</i>	exports : exports object from the node framework, all methods and variables created in the scope of this object will be available in the node program
<i>Return value</i>	<i>none</i>
<i>behavior</i>	After running this method all functions are added to the exports object

For every global function we run

```
NODE_SET_METHOD(exports, functionName, functionProxy)
```

exports is the JavaScript object which we can use to pass methods and data to node

functionName is the name of the function in the current iteration

functionProxy is the function that should be called everytime a class is constructed via JavaScript

All functions are now available using the exports object.

1.7. functionProxy

<i>Name</i>	<code>NodeApplication::functionProxy(const v8::FunctionCallbackInfo<v8::Value>& args)</code>
<i>Visibility</i>	Public
<i>Parameters</i>	args : arguments passed by node
<i>Return value</i>	<i>none</i> , return value will be passed via the args object
<i>behavior</i>	The passed arguments will be converted to C++ native arguments or root objects, the list of all functions will be checked for a function with the given name. A overloaded version of the function with correct parameters will be searched, and called. The results will be converted to JavaScript objects.
<i>Exceptions</i>	In case there is no function with a matching signature, an exception will be thrown. Exceptions from ROOT will be forwarded

This is possible because we can store data in a secret area that is not visible via JavaScript, in there we will store data describing the function call.

1.8. getListOfVariables

<i>Name</i>	<code>NodeApplication::getListOfVariables()</code>
<i>Visibility</i>	Private
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	TCollection of TGlobal A collection containing all globally available variables in the ROOT framework
<i>behavior</i>	<code>gROOT->GetListOfGlobals</code> will be used to get the TCollection

This method will more or less bypass the results of `gROOT->GetListOfGlobals` and is only there if we run into a situation where we need to manipulate the collection before exporting the variables.

1.9. exposeVariables

<i>Name</i>	<code>NodeApplication::exposeVariables(Local<Object> exports)</code>
<i>Visibility</i>	Private
<i>Parameters</i>	exports: exports object from the node framework, all methods and variables created in the scope of this object will be available in the node program
<i>Return value</i>	<i>none</i>
<i>behavior</i>	After running this method all global variables are added to the exports object, by creating a getter and a setter function

For every global variable we create an `ObjectTemplate` and use `v8::Template::SetNativeDataProperty` to set getters and setters for the variable (see `getterProxy` and `setterProxy`). The `Object` generated from the `ObjectTemplate` is being attached to the exports object.

1.10. getterProxy

<i>Name</i>	<code>NodeApplication::getterProxy(const v8::FunctionCallbackInfo<v8::Value>& args)</code>
<i>Visibility</i>	Public
<i>Parameters</i>	args : arguments passed by node, will be ignored
<i>Return value</i>	<i>none</i> , return value will be passed via the args object
<i>behavior</i>	The global variable will be read and converted to a JavaScript object
<i>Exceptions</i>	In case the variable is not readable (e.g. not initialized), null will be returned

We only use one proxy for every setter to minimize overhead. This is possible because we can store data in a secret area that is not visible via JavaScript, in there we will store data describing the getter call.

1.11. setterProxy

<i>Name</i>	<code>NodeApplication::setterProxy(const v8::FunctionCallbackInfo<v8::Value>& args)</code>
<i>Visibility</i>	Public
<i>Parameters</i>	args : first argument will be used
<i>Return value</i>	<i>none</i>
<i>behavior</i>	The parameter in the args object will be converted to a C++ variable, it will be saved in the global variable
<i>Exceptions</i>	In case the variable has a wrong type an Exception will be thrown

We only use one proxy for every setter to minimize overhead. This is possible because we can store data in a secret area that is not visible via JavaScript, in there we will store data describing the setter call.

1.12. additional private methods

- `createGetterName(std::string name)` will change the first character of name to uppercase and prepend get (e.g. name: app, result: getApp)
- `createSetterName(std::string name)` will change the first character of name to uppercase and prepend set (e.g. name: app, result: setApp)

1.13. additional public methods

- A sync method might be needed when new classes can be created during runtime, **need to evaluate this!**

2. ProxyObjectFactory

The ProxyObjectFactory is used whenever a v8 objects needs to be converted into it's ROOT counterpart or vice versa. In a lot of cases we have a memory addresses and types, described by a string. The ProxyObjectFactory needs to parse the type string in order to decide on a class (these classes shall be called ProxyObjects) to which the void type variable needs to be forwarded. These ProxyObjects cotain the correct semantics to generate ROOT and Node objects. They are all named [type]ProxyObject derive from the class ProxyObject. The following ProxyObjects need to implemented to make the bindings work:

- **String**ProxyObject
- **Number**ProxyObject
- **Bool**ProxyObject
- **Object**ProxyObject
- ...

2.1. createProxyObject

<i>Name</i>	ProxyObjectFactory::createProxyObject(void* obj, std::string type))
<i>Visibility</i>	Public static
<i>Parameters</i>	obj : The object to be converted type : The typename, provided by ROOT
<i>Return value</i>	ProxyObject The ProxyObjects that matches the given type. The ProxyObject has already been initialized with the obj pointer.
<i>behavior</i>	Decides which ProxyObject can handle the obj pointer and returns an instance of this

Before creating a new ProxyObject this should query the ProxyObjectCache, checking if the object at the given momeory address has already been proxied.

3. ProxyObject

ProxyObject is an interface defining the following abstract methods:

3.1. isScalar

<i>Name</i>	<code>ProxyObject::isScalar()</code>
<i>Visibility</i>	Public abstract
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	bool true: The object is scalar, no recursion is needed to create a ROOT/v8 representation
<i>behavior</i>	This is usually just a return statement, as String, Number, Bool, ... ProxyObjects will always handle scalar data and ObjectProxyObjects will be the only non scalar ProxyObjects (and will therefor return false)

3.2. getV8Handle

<i>Name</i>	<code>ProxyObject::getV8Handle()</code>
<i>Visibility</i>	Public abstract
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	v8::Handle A Handle will be generated containing the data, used to initialize the ProxyObject.
<i>behavior</i>	This highly depends on the object's type scalar types might just call a constructor and return the result, ObjectProxyObjects will need to step down all through the objects children.

4. ProxyObjectCache

The ProxyObjectCache is a key value store, mapping memory addresses to V8 or ROOT objects.

Some sort of cache is essential in this case, because we might need to deal with cyclic dependencies which would lead to an endless recursion in the ProxyObjectFactory. Using a cache would make cyclic references easy to handle. During conversion of an object it's content must not change so that the cache is valid, after the conversion, the whole cache should be considered dirty because we cannot monitor all changes in either ROOT's nor node's data.

Invalidating the cache after each conversion means that there is more or less no performance gain, in most cases, when we only have linear references, there are only objects written to the cache without any cache hit before invalidation.

IDEA: Make the cache locally instead of globally available, so that instead of invalidate it, we reinitialize it.

4.1. invalidate

<i>Name</i>	ProxyObjectCache::invalidate()
<i>Visibility</i>	Public
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	<i>none</i>
<i>behavior</i>	Clears (or invalidates) the whole cache, so that a cache query does not have a hit, even if it has been mapped before the invalidate call

4.2. put

<i>Name</i>	<code>ProxyObjectCache::put(long key, void value)</code>
<i>Visibility</i>	Public
<i>Parameters</i>	key: is the key which should refer to value, value: is the value to be cached
<i>Return value</i>	<i>none</i>
<i>behavior</i>	Stores the value in the cache and makes it available so that <code>value == cache->get(key)</code> is true after this call

4.3. get

<i>Name</i>	<code>ProxyObjectCache::get(long key)</code>
<i>Visibility</i>	Public
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	void: the value mapped to the given key
<i>behavior</i>	Returns the value that is related to the given key, if the entry is still valid

5. Appendix

5.1. Glossary