Karlsruhe Institute of Technology

Software Engineering Practice

WINTER TERM 2015/2016

rootJS - module guide

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 this document

This document describes the structure of RootJS, it will be used as a blueprint in the implementation phase.

The document contains descriptions to all public methods, so that the implementation can be split up, it contains all classes that need to be implemented, descriptions to all public methods and, listed in the UML diagram in the appendix, some private functions and properties that might be handy during implementation.

1.2. Overview

When using node moldues to extend the basic node api, one uses the *require* statement, providing the name of the module. require returns a JavaScript object wich is called exports, containing everyting the node modules decides to include.

In our case require will run the *initilize* method which will crawl trough ROOT to find all gloablly accessible variables, functions and classes.

For all these items a property or function is being added to the exports object. These properties are bound to a callback function which is equipped with meta data, refereing to the property or function in ROOT. With this information the callback function is able to call the actual ROOT functionality.

To send the results to node we need to convert the resulting objects or values. In order to convert the data we will use proxys for the different datatypes that will be returned by a factory.

The factory will use the datatype to select a matching Proxy implementation, when dealing with non scalar data, the factory will run through all methods and properties and use the Factory recursively.

Even though node programs are mainly used as server applications it can still handle graphical user interfaces. Graphical user interfaces need to refresh frequently in order to be responsive, as JavaScript only runs one thread at a time, the gui refresh needs to run on the same thread as the rest of the application. Graphical user interfaces are therefor supported as well.

This is handled in the *NodeApplication* class, further we will set the application name and a callback function for messages generated by root here.

2. CallbackHandler

The CallbackHandler class gets invoked whenever an encapsulated ROOT function or object is accessed. The callback functions follow one general pattern, when called from a nodeJS program *CallbackInfo* is provided. In the initialization phase we can save *InternalField*s which are belonging to these *CallbackInfo*s. The internal fields are therefore filled with information about the associated ROOT functionality. The callback function uses this information to determine what to do exactly.

An inheritant of Proxy will be used to access the data or call the function / constructor and generate a nodeJS representation of the value to be returned.

2.1. ctorCallback

Name	`CallbackHandler::ctorCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo<Value>* information about the context
Return value	**none**
Behavior	*Whenever the last parameter passed via JavaScript is a JavaScript function, it will be handled as a callback. The following will be done in a new thread, the result is then beeing passed as a parameter when the callback is being called.* This method gets invoked whenever a constructor function of an encapsulated ROOT class is being called. This method should decide which constructor should be invoked, by checking constructor overloads. This constructor needs to be called and the resultuing object needs to be forwarded to the ProxyObjectFactory in order to Proxy the results.

2.2. memberGetterCallback

Name	`CallbackHandler::memberGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, info: PropertyCallbackInfo<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated (class) member was requested. The function will not be mapped to a JavaScript function, but to a getter that is being invoked whenever a variable is requested. Therefore we do not need to provide the ability to use callbacks here, they could not be passed. In addition to that, it should not take long to read a variable.

2.3. memberSetterCallback

Name	`CallbackHandler::memberSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, value: Local<Value>, info: PropertyCallback-Info<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated (class) member is attempted to be set. The function will not be mapped to a JavaScript function but to a setter that is being invoked whenever a variable is saved using the =̈öperator. Therefore we do not need to provide the ability to use callbacks here, they could not be passed. In addition to that, it should not take long to write to a variable.

2.4. memberFunctionCallback

Name	`CallbackHandler::memberFunctionCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo<Value>*
Return value	**none**
Behavior	*Whenever the last parameter passed via JavaScript is a JavaScript function, it will be handled as a callback. The following will be done in a new thread, the result is then beeing passed as a parameter when the callback is being called.* This method gets invoked whenever a method is called. First a method with the correct signature is selected. The selected method will be called and the result will be send trough the ProxyObjectFactory.

2.5. staticGetterCallback

Name	`CallbackHandler::staticGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, info: PropertyCallbackInfo<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated static property was requested. The function will not be mapped to a JavaScript function, but to a getter that is being invoked whenever a variable is requested. Therefore we do not need to provide the ability to use callbacks here, they could not be passed. In addition to that, it should not take long to read a variable.

2.6. staticSetterCallback

Name	`CallbackHandler::staticSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, value: Local<Value>, info: PropertyCallback-Info<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated static property is attempted to be set. The function will not be mapped to a JavaScript function but to a setter that is being invoked whenever a variable is saved using the =öperator. Therefore we do not need to provide the ability to use callbacks here, they could not be passed. In addition to that, it should not take long to write to a variable.

2.7. staticFunctionCallback

Name	`CallbackHandler::staticFunctionCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo<Value>*
Return value	**none**
Behavior	*Whenever the last parameter passed via JavaScript is a JavaScript function, it will be handled as a callback. The following will be done in a new thread, the result is then beeing passed as a parameter when the callback is being called.* This method gets invoked whenever a static method is called. First a method with the correct signature is selected. The selected method will be called and the result will be send trough the ProxyObjectFactory.

3. NodeHandler

The *NodeHandler* is the main entry point when you require RootJS by using

```
// 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)
```

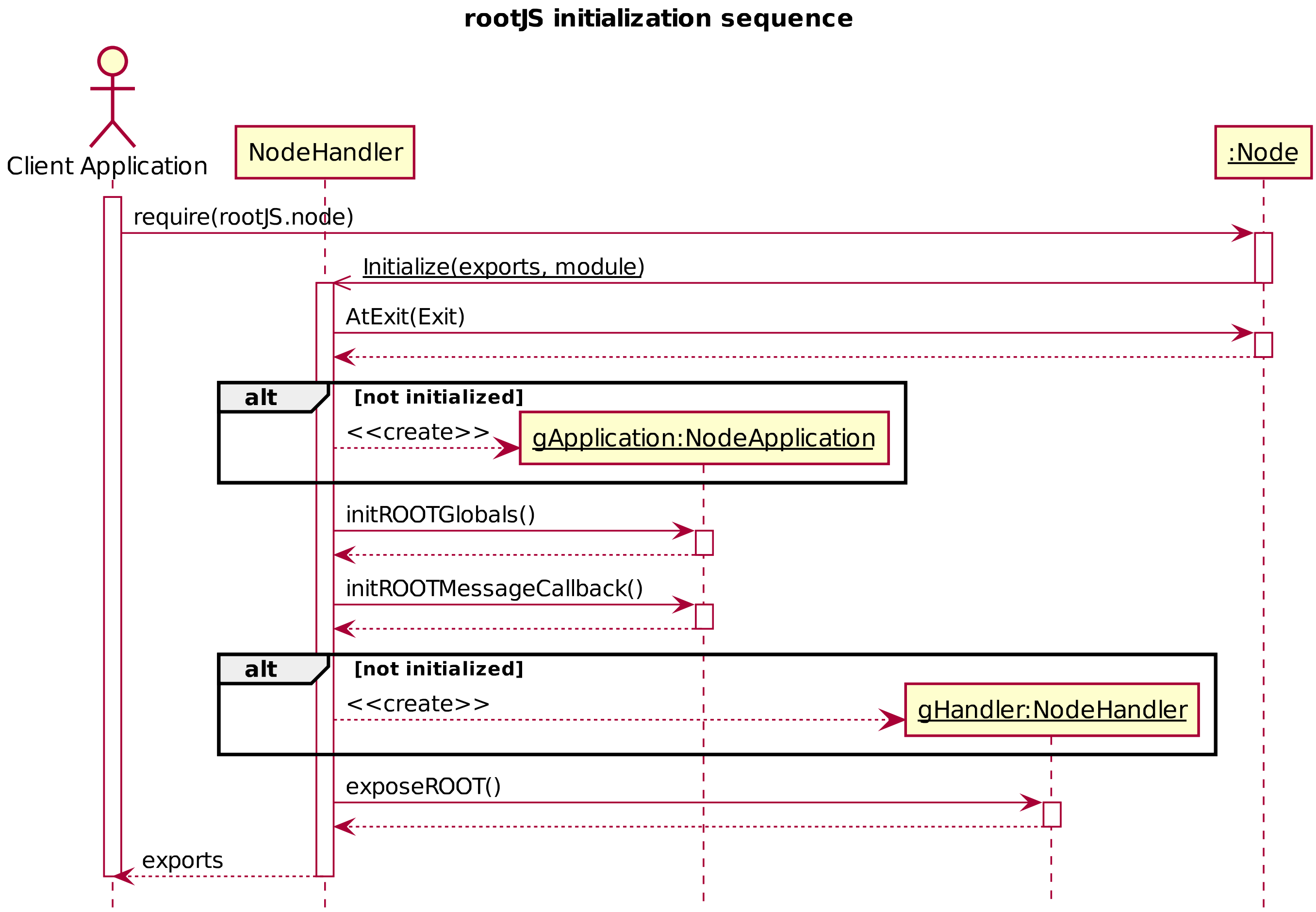

Figure 3.1: initialization sequence

After running the *initialize* method ROOT is fully initialized and all features are exposed to JavaScript.

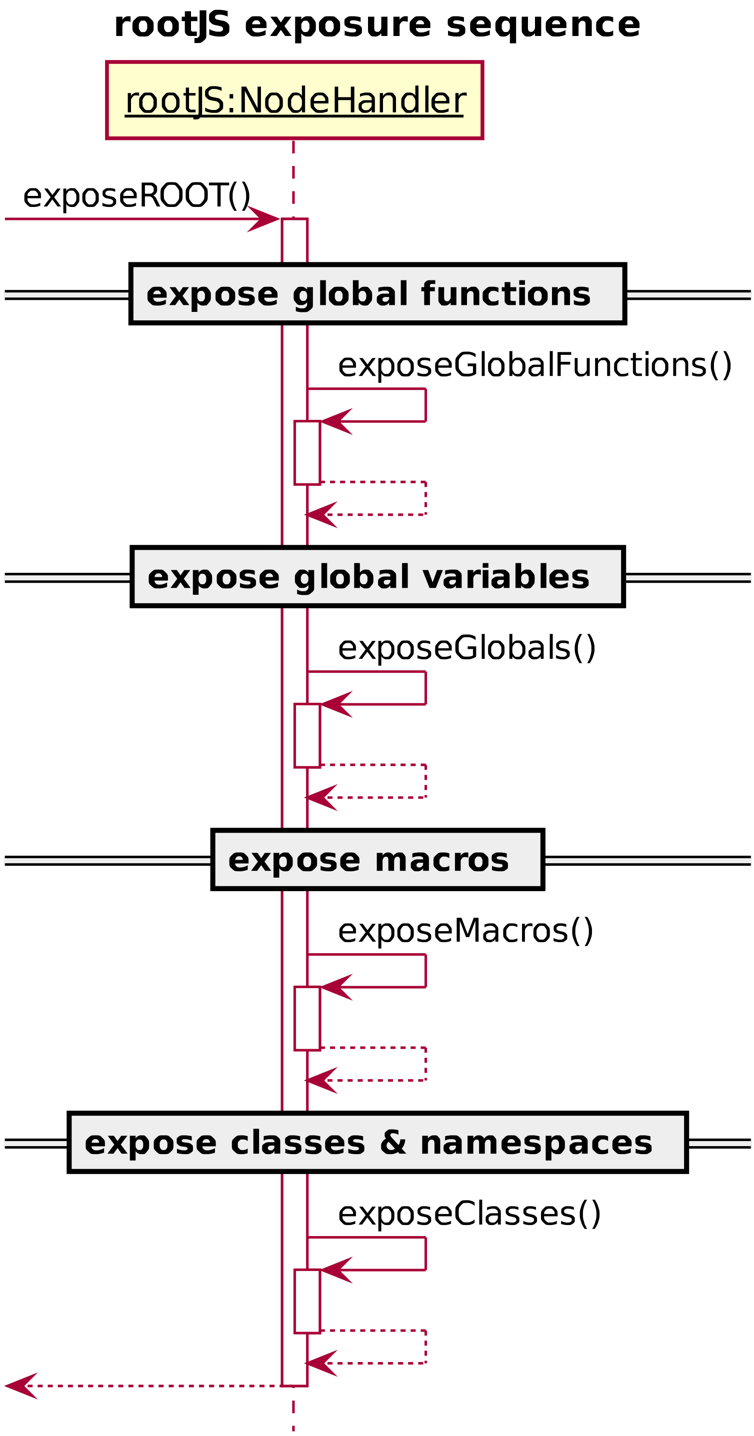

Figure 3.2: class exposure sequence

3.1. initialize

Name	`NodeHandler::initialize(exports: Local<Object>, module: Local<Object>)`
Visibility	public static
Parameters	*exports: Local<Object>, module: Local<Object>* parameters passed by NodeJS
Return value	*none* The features will be exported by passing them to the exports parameter
Behavior	This will create an instance of *NodeApplication* and store it in gApplication, to ensure that all ROOT functionality that relies on gApplication will function properly. Further this will run *getExports* to retrieve the features to be exported to JavaScript which will then be put into the exports object which has been passed to this method

3.2. getExports

Name	`NodeHandler::getExports()`
Visibility	public
Parameters	*none*
Return value	**Local<Object>** features to be exported
Behavior	This method will run multiple private methods to collect global functions, global variables, macros and classes. All these items will be stored in a v8 object which will be passed to RootJS via the initialize method.

4. NodeApplication

ROOT uses TApplication to interface with the windowing system and event handlers. An insatnce of TApplication is usually stored in the global *gApplication* variable.

The main problem with using TApplication directly would be, that we could not hook into the *InitializeGraphics* method. When having a graphical user interface we need to do a UI update frequently:

```
gSystem->ProcessEvents();
```

To avoid having a lot of *ProcessEvents* calls, we wait until *InitializeGraphics* has been called at least once.

Further NodeApplication is being used to set the application's name and initialize a custom message callback which can be used to retrive messages in JavaScript.

4.1. NodeApplication

Name	`NodeApplication::NodeApplication(acn: char*, argc: int*, argv: char**)`
Visibility	public
Parameters	*acn: char*, argc: int*, argv: char***
Return value	**«constructor»** describe return value
Behavior	Set's the application name and constructs a custom message handler

5. TemplateFactory

Creates Javascript function templates from a given ROOT class using *TClassRef*. Methods and static members are set during creation through the use of ROOT reflections and the proxy factories.

5.1. createTemplate

Name	`TemplateFactory::createTemplate(clazz: TClassRef)`
Visibility	public
Parameters	*clazz: TClassRef* the class for which a template is to be created
Return value	**Local<FunctionTemplate>** the created template
Behavior	Gets the class from TClassRef and creates a new function template. Then it iterates over all static members of the class and sets the corresponding members of the template to respective proxy objects. It then iterates through the functions and also sets them. For further reference consider the following sequence diagram.

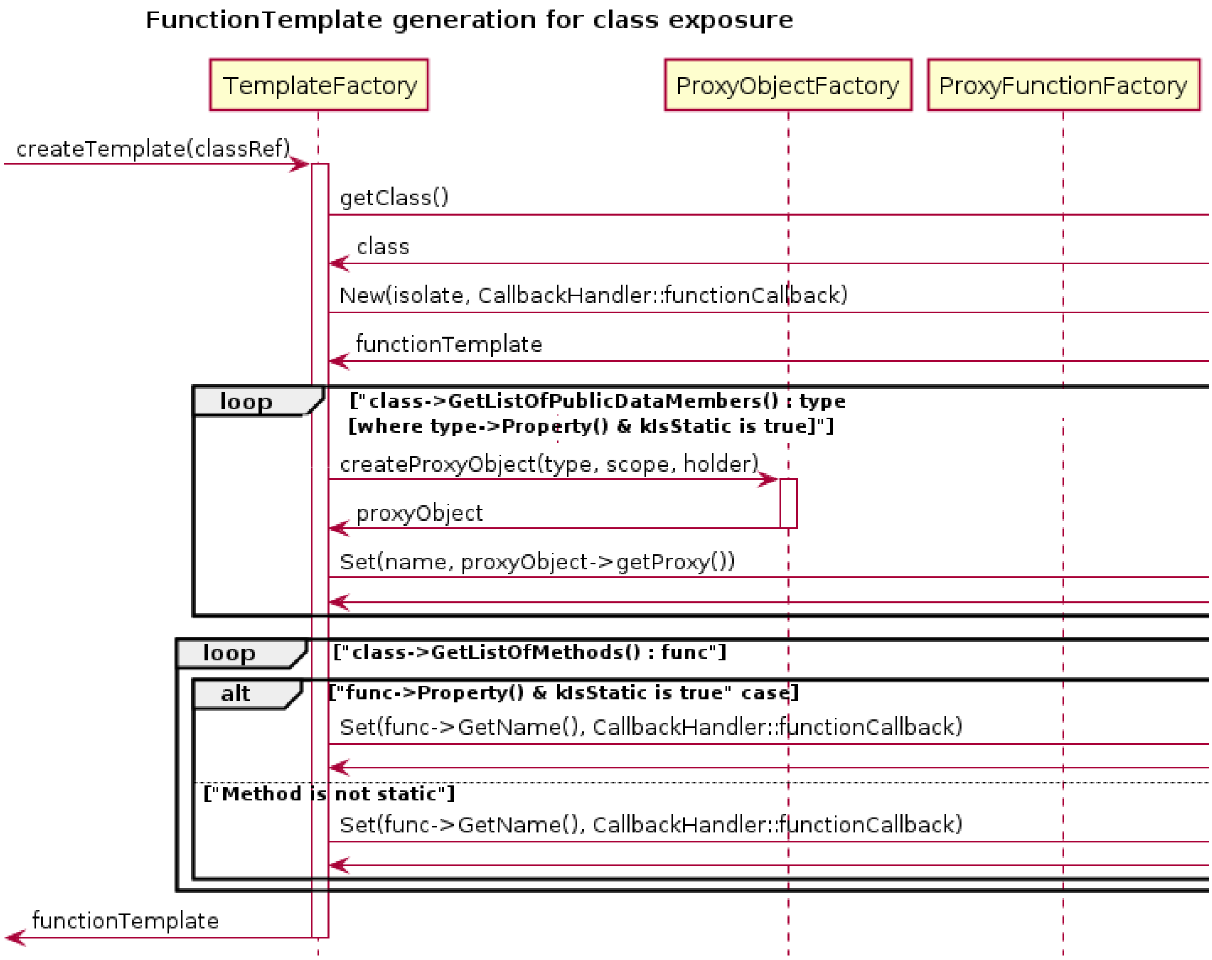

Figure 5.1: function template creation (full diagram in appendix)

6. Proxy

The *Proxy* class is an abstract class which acts as an intermediary between Node.js and ROOT. Both the *ObjectProxy* and *FunctionProxy* inherit the *Proxy* class. Both of them require the object's or function's *void** address to access the original ROOT object. The *TObject* type is more accurately specified in each class which inherits *Proxy*. The *TClassRef* scope is used to access *TClass* and the necessary information about the class. The *Proxy* class holds the data, which both *ObjectProxy* and *FunctionProxy* require.

Node allows us to set a getter and a setter method by calling *v8::ObjectTemplate::SetAccessor*, so whenever a value is assigned to a property, encapsulated by the bindings, a setter will be called. This setter will retrive the *Proxy*, which has been stored in an internal field which is bound to the JavaScript object. The *Proxy* is used to write the new data to the specific memory address, after making sure the property is writable.

6.1. Proxy

Name	`Proxy::Proxy(address: void*, type: TObject, scope: TClassRef)`
Visibility	protected
Parameters	*address: void** The memory address of the ROOT object *type: TObject* The type of Object will be specified in the subclasses *scope: TClassRef* The reference of the TClass so that it can be accessed
Return value	**«constructor»** Returns a Proxy with the given parameters as a variables
Behavior	The Proxy constructor will be inherited by both ObjectProxy and FunctionProxy. The created Proxy will have the parameters as variables.

6.2. setAddress

Name	`Proxy::setAddress(address: void*)`
Visibility	public
Parameters	*address: void** The address to which the proxied ROOT object should be set to
Return value	**none**
Behavior	Sets the address of the proxied ROOT object.

6.3. getAddress

Name	`Proxy::getAddress()`
Visibility	public
Parameters	*none*
Return value	**void*** The current address of the proxied ROOT object
Behavior	Gets the current address of the proxied ROOT object.

6.4. getType

Name	`Proxy::getType()`
Visibility	public
Parameters	*none*
Return value	**TObject** The current type of the proxied ROOT object
Behavior	Gets the current type of the proxied ROOT object.

6.5. getScope

Name	`Proxy::getScope()`
Visibility	public
Parameters	*none*
Return value	**TClassRef** The current scope of the proxied ROOT object
Behavior	Gets the current scope of the proxied ROOT object.

6.6. isGlobal

Name	`Proxy::isGlobal()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the proxied object is global
Behavior	Checks if a global element (not an object member or a static class member) is proxied.

6.7. isTemplate

Name	`Proxy::isTemplate()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the Proxy is a template
Behavior	Checks if the Proxy is a template, which allows using generic types.

6.8. isConst

Name	`Proxy::isConst()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the proxied value is a constant
Behavior	Checks if the proxied value is a constant.

6.9. isStatic

Name	`Proxy::isStatic()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the proxied value is static
Behavior	Checks if the proxied value is static.

7. FunctionProxyFactory

The *FunctionProxyFactory* creates *FunctionProxy* objects. It differentiates between ROOT functions that can be overloaded and those that can't be.

7.1. createFunctionProxy

Name	`FunctionProxyFactory::createFunctionProxy(function: TFunction, scope: TClassRef)`
Visibility	public
Parameters	*function: TFunction* The ROOT function to be proxied. *scope: TClassRef* The scope of the function.
Return value	**FunctionProxy** the proxied function
Behavior	A simple method to create *FunctionProxy* objects with for a given function in a given scope. This is used if there is no overloading or a *TFunction* is given directly.

7.2. fromArgs

Name	`FunctionProxyFactory::fromArgs(name: string, scope: TClassRef, args: FunctionCallbackInfo)`
Visibility	public
Parameters	*name: string* The name of the ROOT function *scope: TClassRef* The reference to the holding class that is searched for the function *args: FunctionCallbackInfo* The arguments read from the *CallbackHandler*
Return value	**FunctionProxy** The proxied ROOT function
Behavior	This method is used to deal with overloaded functions, since JavaScript doesn't support it. It searches the given scope for a function with the given names and arguments and throws an exception if nothing is found.

8. FunctionProxy

In order to make ROOT callables (i.e. functions and methods) dynamically accessible within the Node.js application, they need to be proxied. The *FunctionProxy* provides such functionality, as well as acting as a static cache for commonly used *FunctionProxy* objects.

A *FunctionProxy* instance holds a pointer to the callable's location in main memory, and reflection data, such as the callable's signature. It also provides functionality to validate parameters and encapsulate them within *ObjectProxy* instances. The *FunctionProxy::call* method can then be used to execute the callable using the *ObjectProxy* instances as parameters. The return value is again encapsulated within an *ObjectProxy* and returned to the caller.
As JavaScript does not support overloading, but C++ does, the *FunctionProxy* can be used to statically get all methods with a specified name. The *FunctionProxy* also maintains a static cache which maps callables to their memory location. This is useful for creating new *FunctionProxy instances.*

8.1. getCallFunc

Name	`FunctionProxy::getCallFunc(method: TFunction*)`
Visibility	public
Parameters	*method: TFunction**: pointer to the ROOT function for which a proxy is to be created
Return value	**CallFunc*** a pointer to the CallFunc object provied by cling
Behavior	Gets a pointer to a *CallFunc* object, which encapsulates the provided ROOT function in memory.

8.2. getMethodsFromName

Name	`FunctionProxy::getMethodsFromName(scope: TClassRef, name: string)`
Visibility	public
Parameters	*scope: TClassRef* reference to the class which is checked for methods with the specified name *name: string* name of the overloaded methods which shall be returned
Return value	**vector<TFunction*>** methods that match the specified name
Behavior	Gets a reference to a class and a method name string. It returnes all methods of the class with the specified name. This is needed since JavaScript does not support method overloading.

8.3. FunctionProxy

Name	`FunctionProxy::FunctionProxy(address: void*, function: TFunction, scope: TClassRef)`
Visibility	public
Parameters	*address: void** memory address of the proxied function *function: TFunction* the function's reflection object *scope: TClassRef* the class that the function belogs to
Return value	**«constructor»** the created *FunctionProxy*
Behavior	Creates the *FunctionProxy*.

8.4. getType

Name	`FunctionProxy::getType()`
Visibility	public
Parameters	*none*
Return value	**TFunction** the *TFunction* object that contains the function's reflection data
Behavior	Returns the wrapped function's *TFunction* object. It contains the meta data of its corresponding function.

8.5. validateArgs

Name	`FunctionProxy::validateArgs(args: FunctionCallbackInfo)`
Visibility	public
Parameters	*args: FunctionCallbackInfo* information about the context of the call, including the number and values of arguments
Return value	**ObjectProxy[]** array of the arguments as proxies
Behavior	Checks whether the function is being called with the proper arguments and wraps them in proxies so they can be used by the call method.

8.6. call

Name	`FunctionProxy::call(args: ObjectProxy[])`
Visibility	public
Parameters	*args: ObjectProxy[]* proxies containing arguments for the method
Return value	**ObjectProxy** proxy for the object returned by the called method
Behavior	Calls the actual method in memory using Cling. The argument object proxies' contents are read and given to the called method.

9. ObjectProxyFactory

The *ObjectProxyFactory* creates *ObjectProxy* instances with *TDataMember* type, *TClassRef* scope and *ObjectProxy* holder. It encapsulates ROOT objects recursively for use in Javascript.

To handle circular references we need to maintain a cache of already generated *ProxyObjects*, which will only be valid during object conversion. Whenever an object is cached it will be used instead of creating a new one.

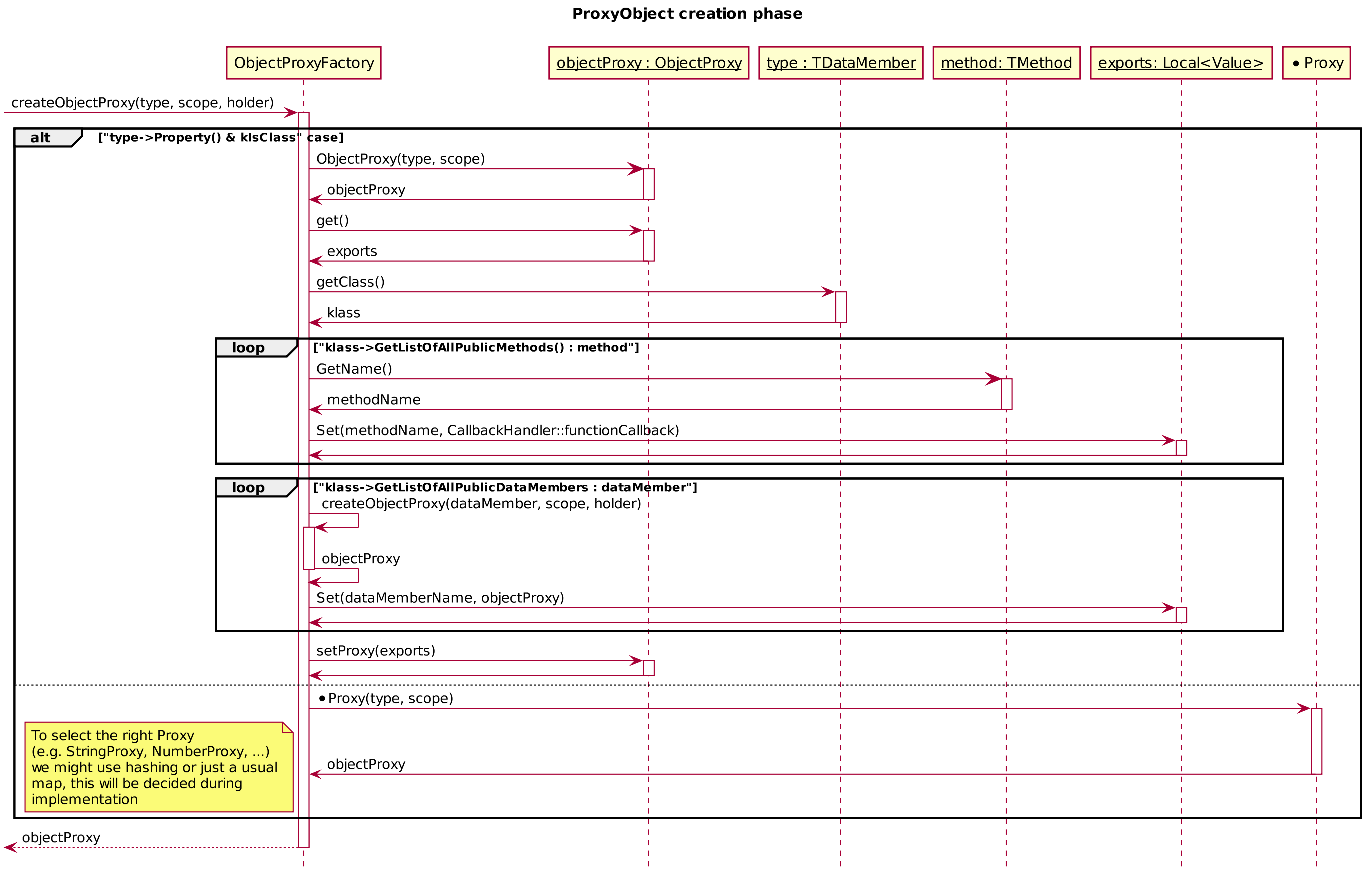

Figure 9.1: object proxy createion sequence

9.1. createObjectProxy

Name	`ObjectProxyFactory::createObjectProxy(type: TDataMember, scope: TClassRef, holder: ObjectProxy)`
Visibility	public
Parameters	*type: TDataMember* The type identification which the ObjectProxy will have *scope: TClassRef* The class the ObjectProxy belongs to *holder: ObjectProxy* The holder is the ObjectProxy which will encapsulate and hold the newly created ObjectProxy
Return value	**ObjectProxy** Returns the ObjectProxy which is created with the given parameters.
Behavior	A new ObjectProxy is created each time the createObjectProxy method is called up.

10. ObjectProxy

The *ObjectProxy* class is used to represent ROOT objects. It differentiates between primitive and non-primitive object types.
There are the following implementations of *ObjectProxy*:

- **EnumProxy** Maps C++ enums to JavaScript strings
- **StructProxy** Maps C++ structs to JavaScript objects
- **ArrayProxy** Maps C++ arrays to JavaScript arrays, we cannot enlarge C++ arrays, so we will throw an Exception on overflows
- **PointerProxy** Maps C++ pointers to JavaScript objects
- **NumberProxy** Uses a C++ template to map all C++ numbers to JavaScript Numbers
- **StringProxy** Maps C++ strings and c-strings to JavaScript strings
- **BooleanProxy** Maps C++ root boolean to Javascript boolean

The *ObjectProxyFactory* decides which *ObjectProxy* needs to be instantiated. Internally all these *ObjectProxies* work the same way by linking a *v8::Local* with a *TDataMember*

10.1. ObjectProxy

Name	`ObjectProxy::ObjectProxy(type: TDataMember, scope: TClassRef)`
Visibility	public
Parameters	*type TDataMember* The type of the object *scope TClassRef* The reference to the class of the object
Return value	**«constructor»** the newly constructed ObjectProxy
Behavior	Creates a new ObjectProxy with the given type and scope.

10.2. getType

Name	`ObjectProxy::getType()`
Visibility	public
Parameters	*none*
Return value	**TDataMember** the type of the ObjectProxy
Behavior	Returns the type of the Object behind the proxy.

10.3. set

Name	`ObjectProxy::set(value: ObjectProxy)`
Visibility	public
Parameters	*value: ObjectProxy* the value to set
Return value	**none**
Behavior	Sets the value of the Object behind the proxy.

10.4. get

Name	`ObjectProxy::get()`
Visibility	public
Parameters	*none*
Return value	**Local<Value>** The value the object has.
Behavior	Returns the value that was set for the object.

10.5. setProxy

Name	`ObjectProxy::setProxy(proxy: Local<Object>)`
Visibility	public
Parameters	*proxy: Local<Object>* The v8 object which will be proxy of the ROOT object.
Return value	**none**
Behavior	Sets *proxy: Local<Object>* to be the proxy of the ROOT object.

10.6. getProxy

Name	`ObjectProxy::getProxy()`
Visibility	public
Parameters	*none*
Return value	**Local<Object>** The current proxy of the ROOT object.
Behavior	Gets the proxy of the ROOT object.

10.7. isPrimitive

Name	`ObjectProxy::isPrimitive()`
Visibility	public
Parameters	*none*
Return value	**bool** Whether or not the represented object is of a primitive type or not.
Behavior	Returns *true* if the represented object's type is primitive, *false* if not.

11. Appendix

11.1. Class diagram

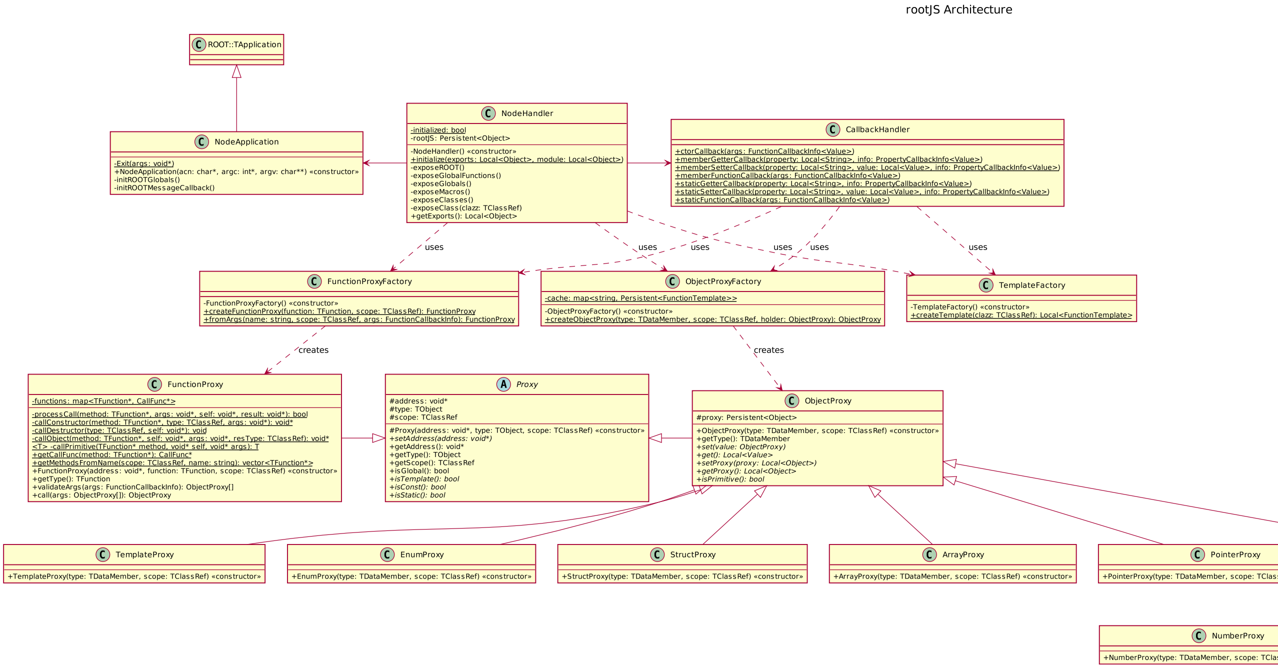

Figure 11.1: rootJS class diagram 1

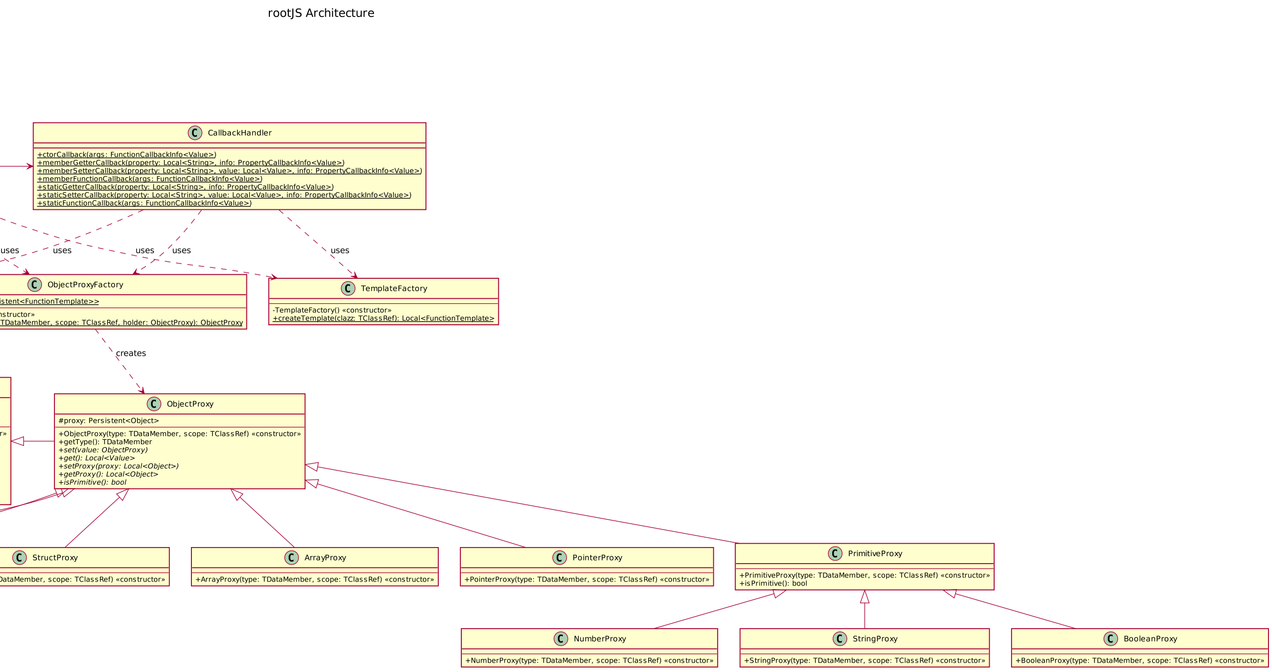

Figure 11.2: rootJS class diagram 2

11.2. Dynamic Model

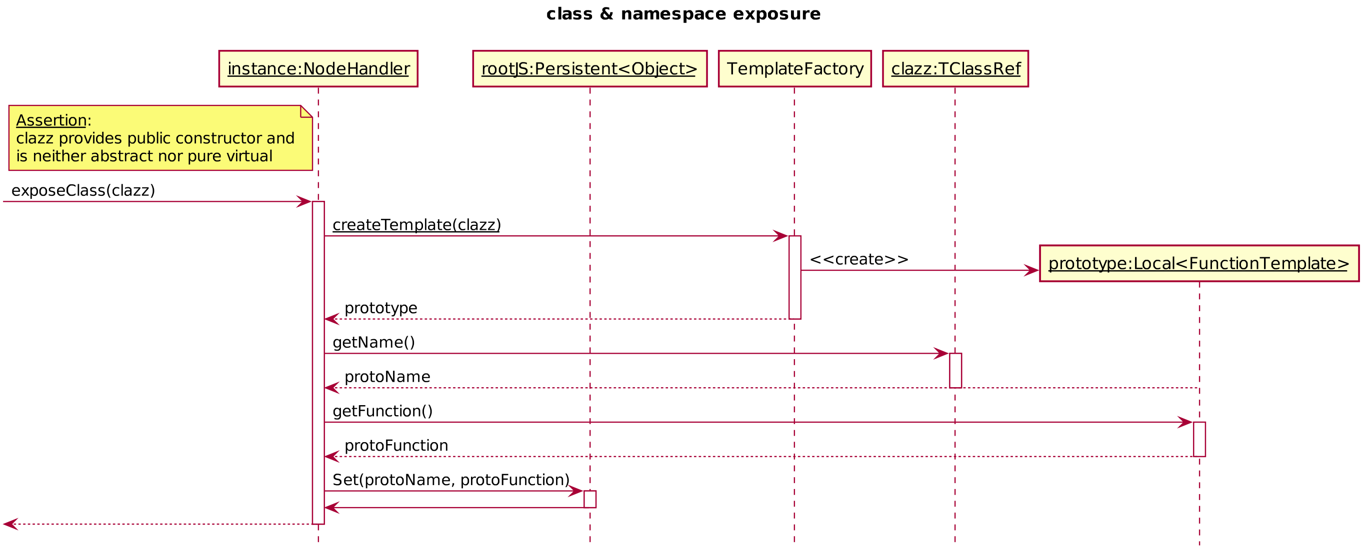

Figure 11.3: class exposure sequence

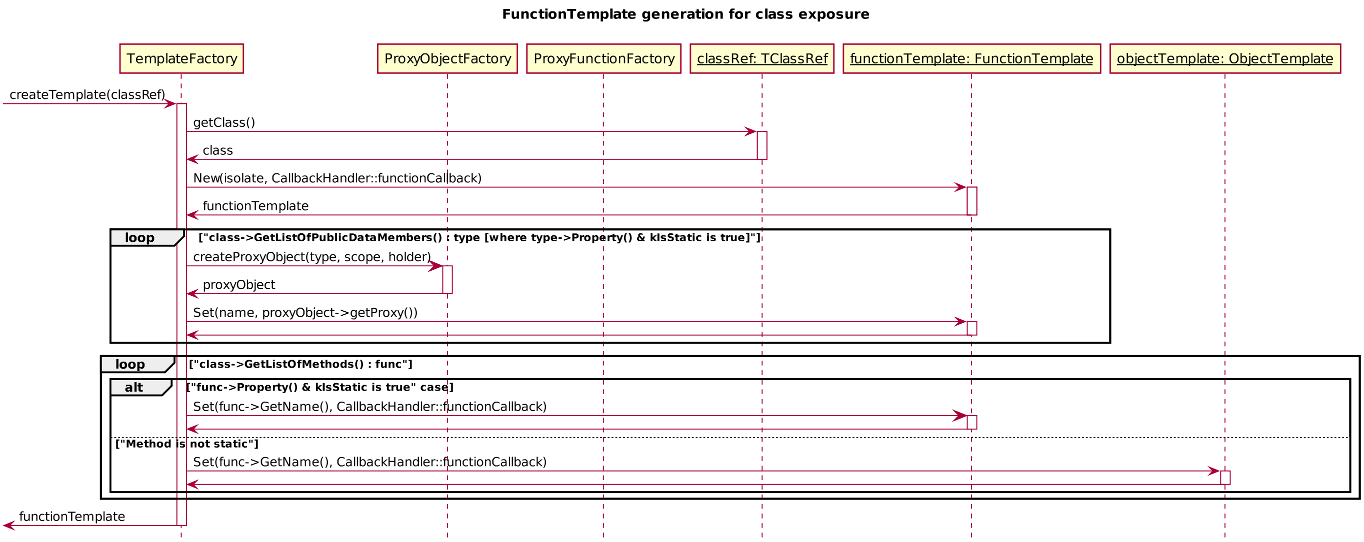

Figure 11.4: class exposure sequence

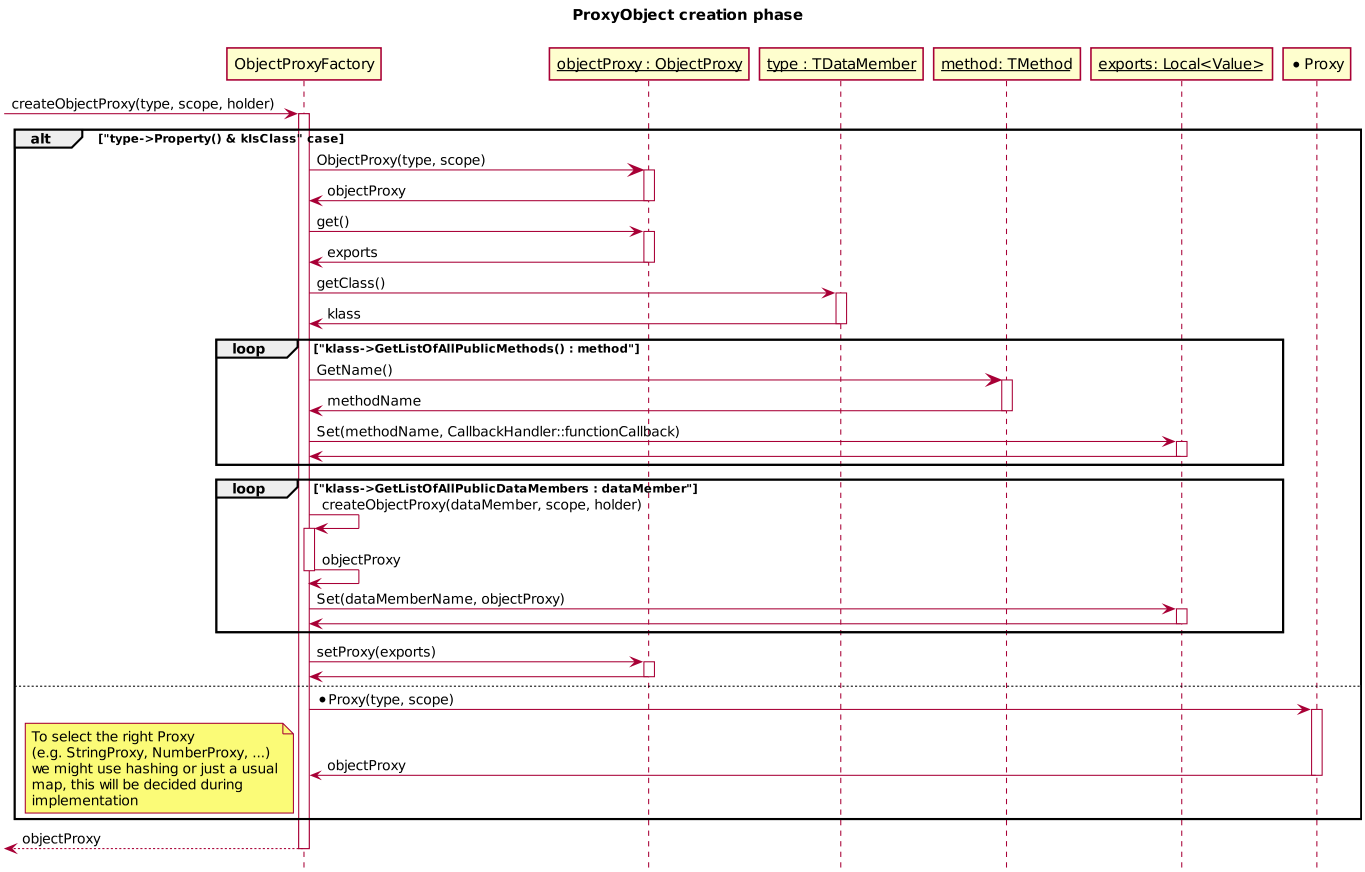

Figure 11.5: ProxyObject creation sequence

11.3. Glossary

Callback

A function which is passed as an argument to some code, which is then expected to call the argument back.

Constructor

A method which is used to create an object.

Encapsulation

A piece of functionality of certain languages used to restrict access to some of the object's variables and methods

Instance

A created object.

Proxy

A class functioning as an intermediary between two classes.

Static

A method which does not require the object to be instantiated.

Template

A feature of C++ that allows classes and functions to operate with generic types.

v8

An open source JavaScript engine, written in C++ and made by Google.