

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	CallbackHandler	3
1.1	ctorCallback	3
1.2	staticCtorCallback	4
1.3	memberGetterCallback	5
1.4	memberSetterCallback	6
1.5	memberFunctionCallback	7
1.6	staticGetterCallback	8
1.7	staticSetterCallback	9
1.8	staticFunctionCallback	10
2	NodeHandler	11
2.1	initialize	11
2.2	getExports	11
3	NodeApplication	12
3.1	NodeApplication	12
4	TemplateFactory	13
4.1	createTemplate	13
5	Proxy	15
5.1	Proxy	15
5.2	setAddress	16
5.3	getAddress	17
5.4	getType	18
5.5	getScope	19
5.6	isGlobal	20
5.7	isTemplate	21
5.8	isConst	22
5.9	isStatic	23
6	FunctionProxyFactory	24
6.1	createFunctionProxy	24
6.2	fromArgs	25
7	FunctionProxy	26
7.1	getCallFunc	26
7.2	getMethodsFromName	27
7.3	FunctionProxy	28
7.4	getType	29
7.5	validateArgs	30
7.6	call	31
8	ObjectProxyFactory	32
8.1	createObjectProxy	32
9	ObjectProxy	33
9.1	ObjectProxy	33
9.2	getType	34
9.3	set	35
9.4	get	36
9.5	setProxy	37
9.6	getProxy	38

9.7 isPrimitive	39
10 Appendix	40
10.1 Class diagram	40
10.2 Dynamic Model	42
10.3 Glossary	43

1. CallbackHandler

The CallbackHandler class gets invoked whenever an encapsulated ROOT function or object is accessed.

1.1. ctorCallback

<i>Name</i>	<code>CallbackHandler::ctorCallback(args: FunctionCallbackInfo<Value>)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>args: FunctionCallbackInfo<Value></i> information about the context
<i>Return value</i>	none
<i>Behavior</i>	Gets invoked whenever a non static constructor function of an encapsulated ROOT class was called.

1.2. staticCtorCallback

<i>Name</i>	<code>CallbackHandler::staticCtorCallback(args: FunctionCallbackInfo<Value>)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>args: FunctionCallbackInfo<Value></i>
<i>Return value</i>	none
<i>Behavior</i>	Gets invoked whenever a static constructor of an encapsulated ROOT class was called.

1.3. memberGetterCallback

<i>Name</i>	<code>CallbackHandler::memberGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)</code>
<i>Visibility</i>	<code>public</code>
<i>Parameters</i>	<code>property: Local<String>, info: PropertyCallbackInfo<Value></code>
<i>Return value</i>	<code>none</code>
<i>Behavior</i>	Gets invoked whenever an encapsulated (class) member was requested.

1.4. memberSetterCallback

<i>Name</i>	<code>CallbackHandler::memberSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)</code>
<i>Visibility</i>	<code>public</code>
<i>Parameters</i>	<code>property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value></code>
<i>Return value</i>	none
<i>Behavior</i>	Gets invoked whenever an encapsulated (class) member is attempted to be set.

1.5. memberFunctionCallback

<i>Name</i>	<code>CallbackHandler::memberFunctionCallback(args: FunctionCallbackInfo<Value>)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>args: FunctionCallbackInfo<Value></i>
<i>Return value</i>	none
<i>Behavior</i>	Gets invoked whenever an non-static (class) function was called.

1.6. staticGetterCallback

<i>Name</i>	<code>CallbackHandler::staticGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)</code>
<i>Visibility</i>	<code>public</code>
<i>Parameters</i>	<code>property: Local<String>, info: PropertyCallbackInfo<Value></code>
<i>Return value</i>	<code>none</code>
<i>Behavior</i>	Gets invoked whenever an encapsulated static object was requested.

1.7. staticSetterCallback

<i>Name</i>	<code>CallbackHandler::staticSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)</code>
<i>Visibility</i>	<code>public</code>
<i>Parameters</i>	<code>property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value></code>
<i>Return value</i>	none
<i>Behavior</i>	Gets invoked whenever an encapsulated static object is attempted to be set.

1.8. staticFunctionCallback

<i>Name</i>	<code>CallbackHandler::staticFunctionCallback(args: FunctionCallbackInfo<Value>)</code>
<i>Visibility</i>	<code>public</code>
<i>Parameters</i>	<code>args: FunctionCallbackInfo<Value></code>
<i>Return value</i>	<code>none</code>
<i>Behavior</i>	Gets invoked whenever a static function was called.

2. 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)
```

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

2.1. initialize

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

2.2. getExports

<i>Name</i>	<code>NodeHandler::getExports()</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	Local<Object> features to be exported
<i>Behavior</i>	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.

3. NodeApplication

ROOT uses TApplication to interface with the windowing system and event handlers. An instance 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 retrieve messages in JavaScript.

3.1. NodeApplication

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

4. 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. The created templates are kept in a cache to avoid unnecessary creation of already existing templates.

4.1. createTemplate

<i>Name</i>	<code>TemplateFactory::createTemplate(clazz: TClassRef)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>clazz</i> : <i>TClassRef</i> the class for which a template is to be created
<i>Return value</i>	Local<FunctionTemplate> the created template
<i>Behavior</i>	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.

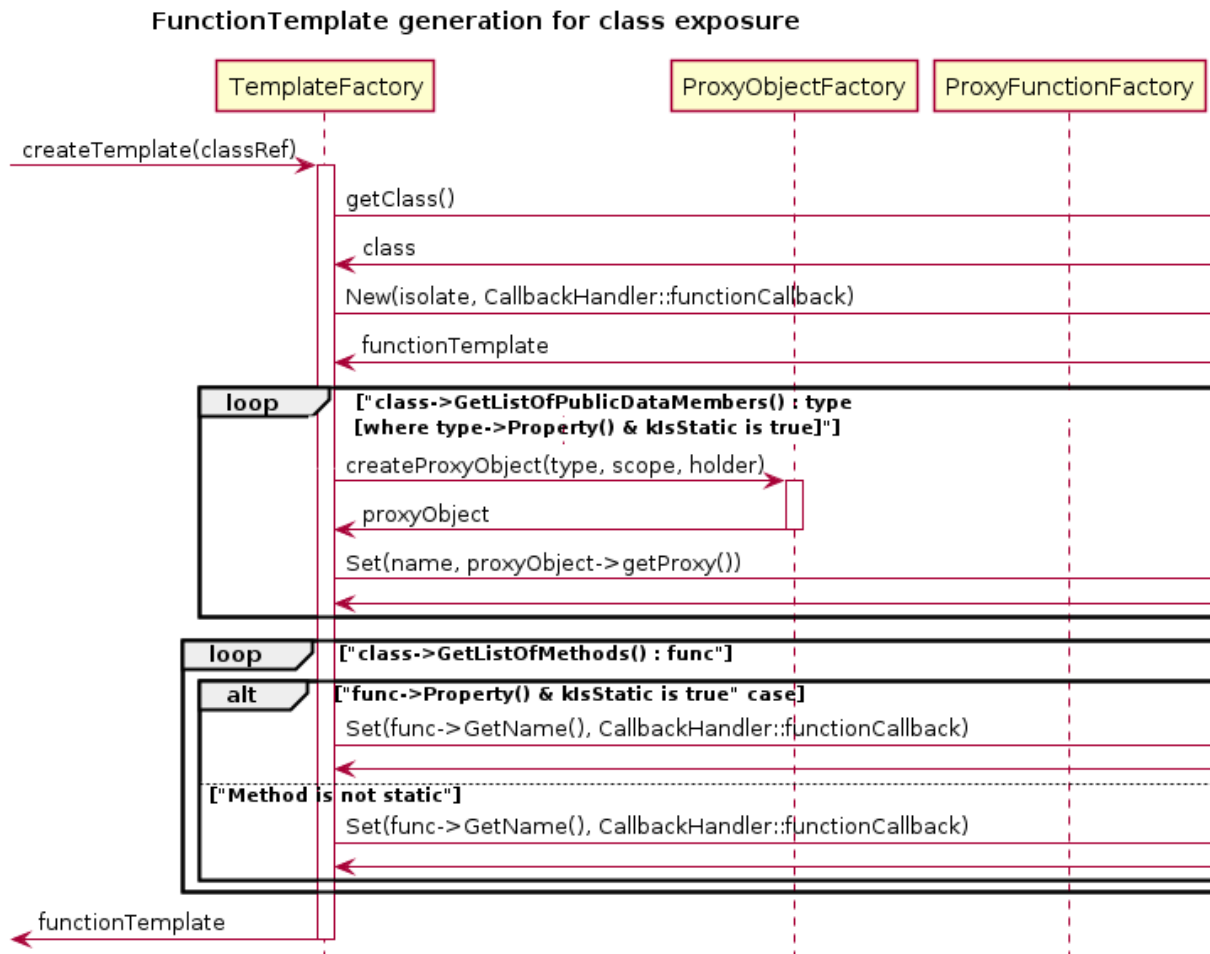


Figure 4.1: function template creation (full diagram in appendix)

5. 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, since both of them require the object's or function's *void** address, *TObject* type and *TClassRef* scope. The *Proxy* class holds the data, which both *ObjectProxy* and *FunctionProxy* require. The *Proxy* class uses the Proxy design pattern.

5.1. Proxy

<i>Name</i>	<code>Proxy::Proxy(address: void*, type: TObject, scope: TClassRef)</code>
<i>Visibility</i>	protected
<i>Parameters</i>	<i>address: void*</i> The memory address of the Proxy <i>type: TObject</i> The type identification which the ObjectProxy will have <i>scope: TClassRef</i> The class the Proxy belongs to
<i>Return value</i>	« constructor » Returns a Proxy with the given parameters as a variables
<i>Behavior</i>	The Proxy constructor will be inherited by both ObjectProxy and FunctionProxy. The created Proxy will have the parameters as variables.

5.2. setAddress

<i>Name</i>	<code>Proxy::setAddress(address: void*)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>address: void*</i> The address to which the Proxy should be set to
<i>Return value</i>	none
<i>Behavior</i>	Sets the address of the Proxy.

5.3. getAddress

<i>Name</i>	<code>Proxy::getAddress()</code>
<i>Visibility</i>	<code>public</code>
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	void* The current address of the Proxy
<i>Behavior</i>	Gets the current address of the Proxy.

5.4. getType

<i>Name</i>	<code>Proxy::getType()</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	TObject The current type of the Proxy
<i>Behavior</i>	Gets the current type of the Proxy.

5.5. getScope

<i>Name</i>	<code>Proxy::getScope()</code>
<i>Visibility</i>	public
<i>Parameters</i>	none
<i>Return value</i>	TClassRef The current scope of the Proxy
<i>Behavior</i>	Gets the current scope of the Proxy.

5.6. isGlobal

<i>Name</i>	<code>Proxy::isGlobal()</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	bool True if the Proxy is global
<i>Behavior</i>	Checks if the Proxy is global and hence visible throughout the program.

5.7. isTemplate

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

5.8. isConst

<i>Name</i>	<code>Proxy::isConst()</code>
<i>Visibility</i>	public
<i>Parameters</i>	none
<i>Return value</i>	bool True if the Proxy is a constant
<i>Behavior</i>	Checks if the Proxy is a constant.

5.9. isStatic

<i>Name</i>	<code>Proxy::isStatic()</code>
<i>Visibility</i>	public
<i>Parameters</i>	none
<i>Return value</i>	bool True if the Proxy is static
<i>Behavior</i>	Checks if the Proxy is static.

6. FunctionProxyFactory

The *FunctionProxyFactory*

6.1. createFunctionProxy

<i>Name</i>	<code>FunctionProxyFactory::createFunctionProxy(function: TFunction, scope: TClassRef)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>function: TFunction</i> The ROOT function to be proxied. <i>scope: TClassRef</i>
<i>Return value</i>	ProxyFunction the proxied function
<i>Behavior</i>	Creates <i>FunctionProxy</i> objects with <i>TFunction</i> function and <i>TClassRef</i> scope.

6.2. fromArgs

<i>Name</i>	<code>FunctionProxyFactory::fromArgs(name: string, scope: TClassRef, args: FunctionCallbackInfo)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>name: string, scope: TClassRef, args: FunctionCallbackInfo</i> The name of the function, its scope and the arguments it takes.
<i>Return value</i>	FunctionProxy describe return value
<i>Behavior</i>	This method is called from the <i>createFunctionProxy</i> method to deal with overloaded functions, since JavaScript doesn't support it.

7. FunctionProxy

Acts as a proxy for a ROOT callable (i.e. function or class method). It provides methods to execute such a callable and validate its arguments. It also maintains a map of *TFunction* - *CallFunc* entries to cache already used functions.

7.1. getCallFunc

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

7.2. getMethodsFromName

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

7.3. FunctionProxy

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

7.4. getType

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

7.5. validateArgs

<i>Name</i>	<code>FunctionProxy::validateArgs(args: FunctionCallbackInfo)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>args</i> : <i>FunctionCallbackInfo</i> information about the context of the call, including the number and values of arguments
<i>Return value</i>	ObjectProxy[] array of the arguments as proxies
<i>Behavior</i>	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.

7.6. call

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

8. ObjectProxyFactory

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

8.1. createObjectProxy

<i>Name</i>	<code>ObjectProxyFactory::createObjectProxy(type: TDataMember, scope: TClassRef, holder: ObjectProxy)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<p><i>type: TDataMember</i> The type identification which the ObjectProxy will have</p> <p><i>scope: TClassRef</i> The class the ObjectProxy belongs to</p> <p><i>holder: ObjectProxy</i> The holder is the ObjectProxy which will encapsulate and hold the newly created ObjectProxy</p>
<i>Return value</i>	ObjectProxy Returns the ObjectProxy which is created with the given parameters. 1q
<i>Behavior</i>	A new ObjectProxy is created each time the createObjectProxy method is called up.

9. 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*

9.1. ObjectProxy

<i>Name</i>	<code>ObjectProxy::ObjectProxy(type: TDataMember, scope: TClassRef)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>type: TDataMember, scope: TClassRef</i> the type and scope of the object
<i>Return value</i>	« constructor » the newly constructed ObjectProxy
<i>Behavior</i>	Creates a new ObjectProxy with the given type and scope.

9.2. getType

<i>Name</i>	<code>ObjectProxy::getType()</code>
<i>Visibility</i>	public
<i>Parameters</i>	none
<i>Return value</i>	TDataMember the type of the ObjectProxy
<i>Behavior</i>	Returns the type of the Object behind the proxy.

9.3. set

<i>Name</i>	<code>ObjectProxy::set(value: ObjectProxy)</code>
<i>Visibility</i>	<code>public</code>
<i>Parameters</i>	<i>value: ObjectProxy</i> the value to set
<i>Return value</i>	none
<i>Behavior</i>	Sets the value of the Object behind the proxy.

9.4. get

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

9.5. setProxy

<i>Name</i>	<code>ObjectProxy::setProxy(proxy: Local<Object>)</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>proxy: Local<Object></i>
<i>Return value</i>	none
<i>Behavior</i>	describe beahviour

9.6. getProxy

<i>Name</i>	<code>ObjectProxy::getProxy()</code>
<i>Visibility</i>	public
<i>Parameters</i>	<i>none</i>
<i>Return value</i>	Local<Object> describe return value
<i>Behavior</i>	describe beahviour

9.7. isPrimitive

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

10. Appendix

10.1. Class diagram

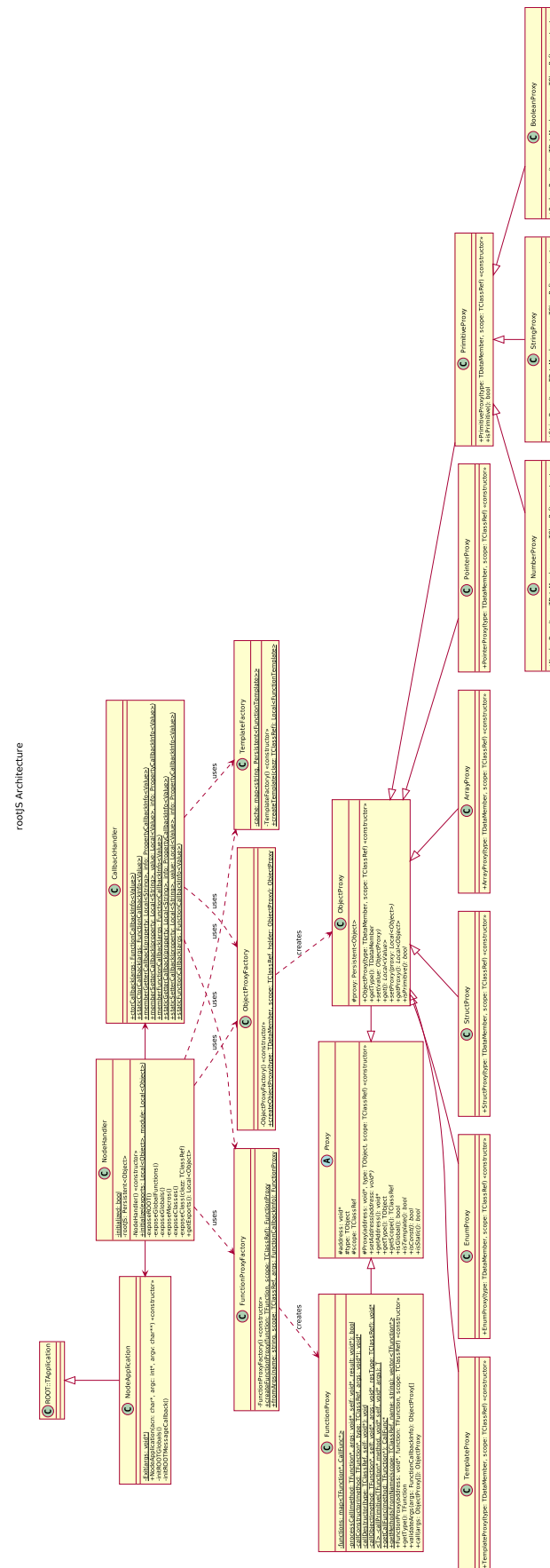


Figure 10.1: rootJS class diagram

10.2. Dynamic Model

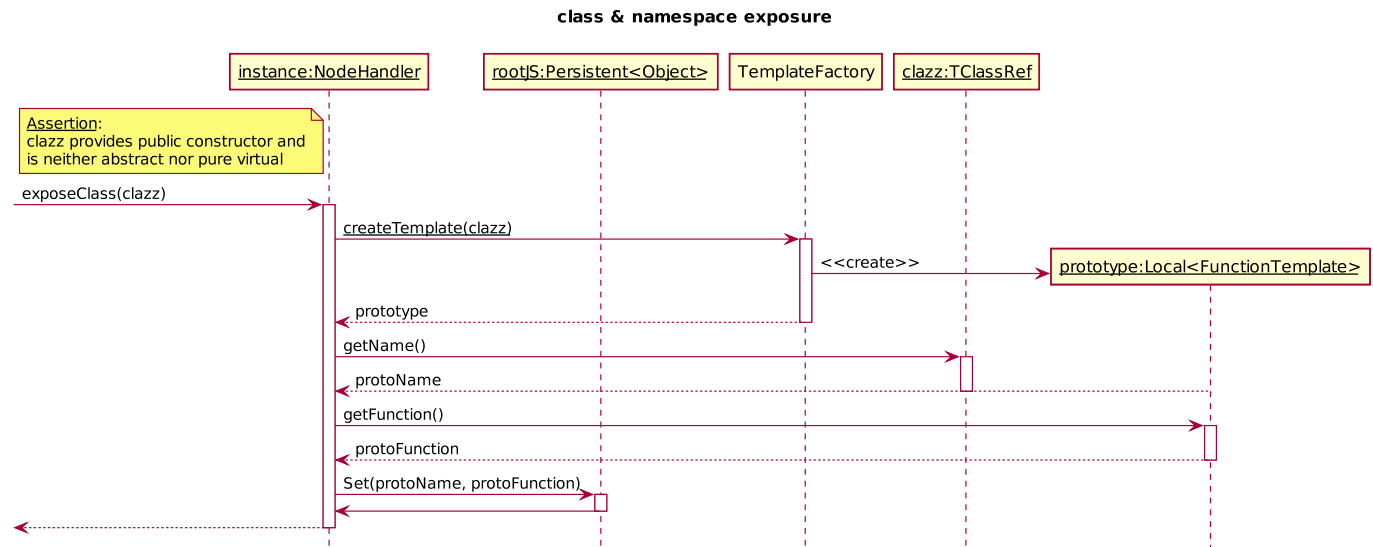


Figure 10.2: class exposure sequence

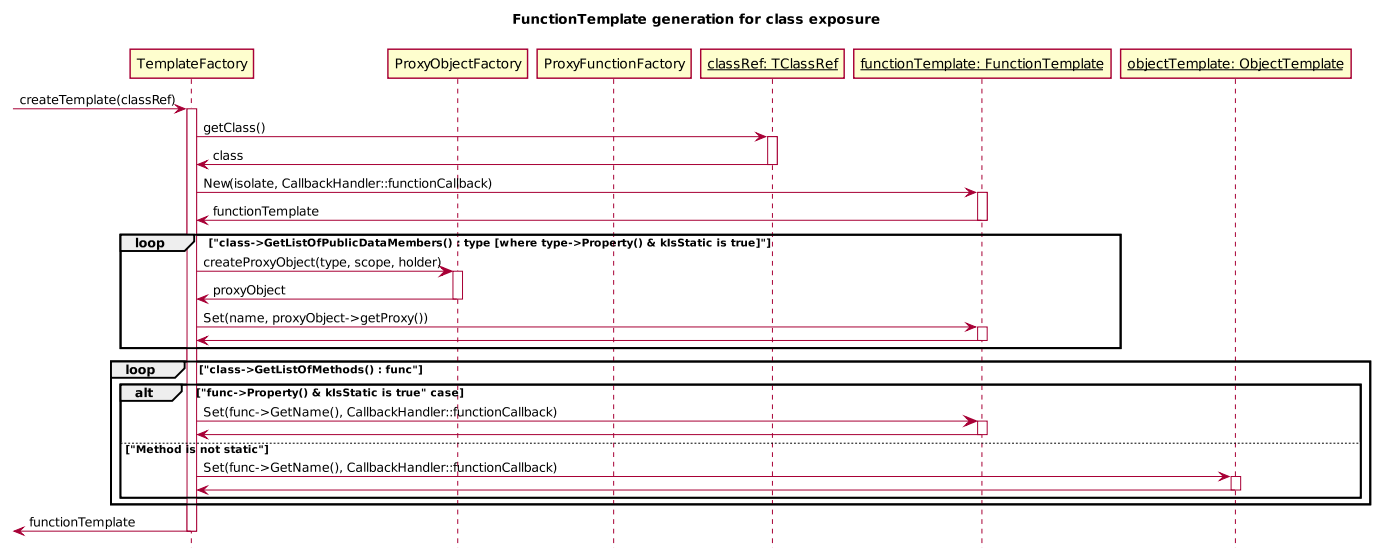


Figure 10.3: class exposure sequence

10.3. Glossary