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. 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.

1.1. ctorCallback

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

1.2. staticCtorCallback

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

1.3. 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.

1.4. memberSetterCallback

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

1.5. memberFunctionCallback

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

1.6. 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 object was requested.

1.7. staticSetterCallback

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

1.8. staticFunctionCallback

Name	`CallbackHandler::staticFunctionCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo<Value>*
Return value	**none**
Behavior	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

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

2.2. 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.

3. 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.

3.1. 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

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

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.

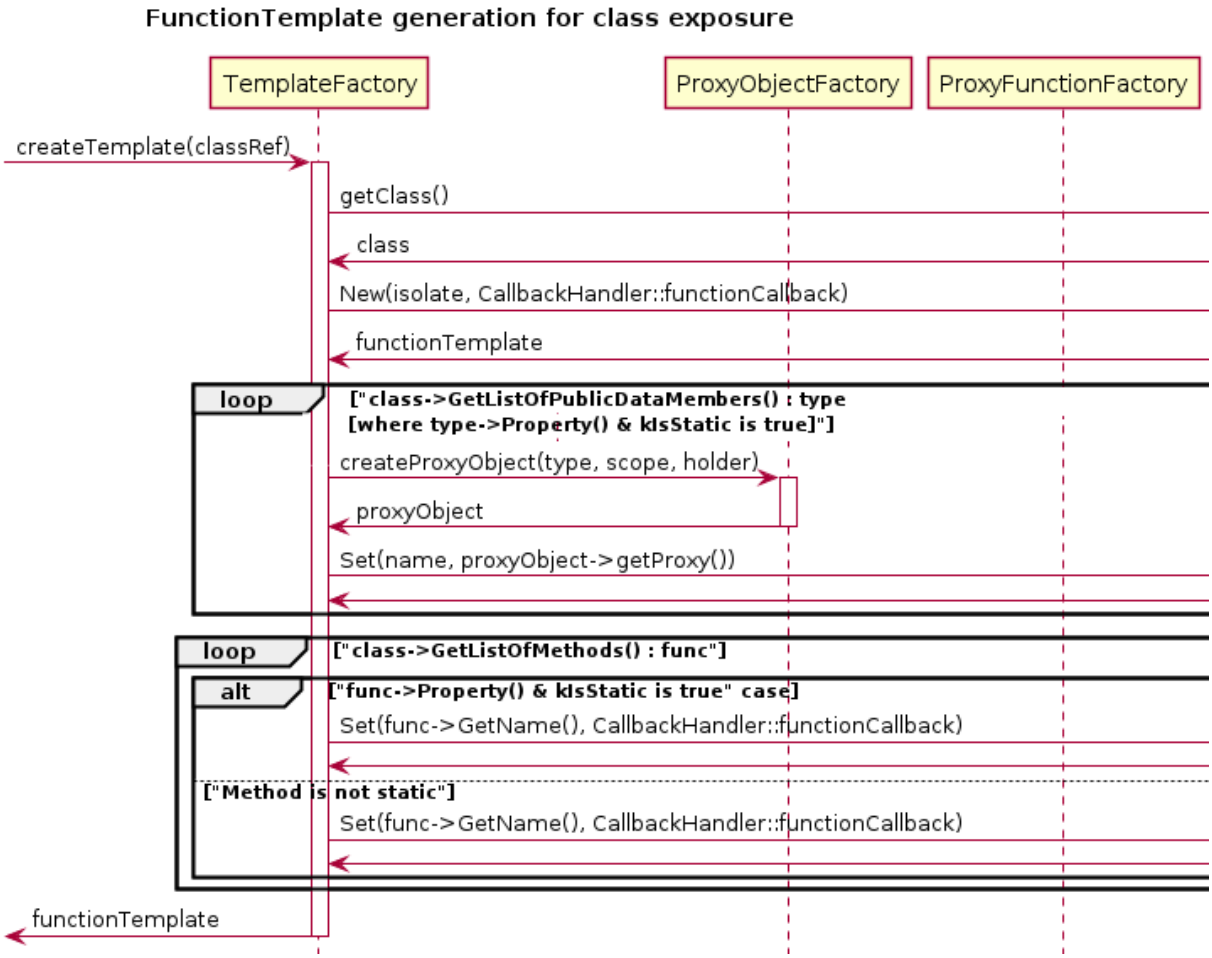

Figure 4.1: function template creation (full diagram in appendix)