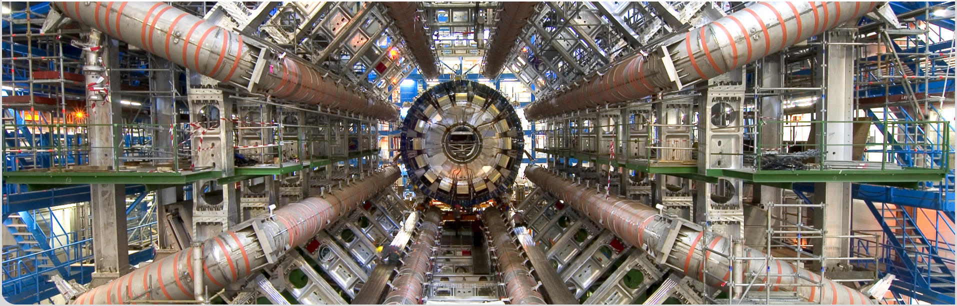
rootJS - Functional Specification

PSE - Software Engineering Practice

C. Wolff, M. Früh, S. Rajgopal, C. Haas, J. Schwabe, T. Beffart | December 16, 2015

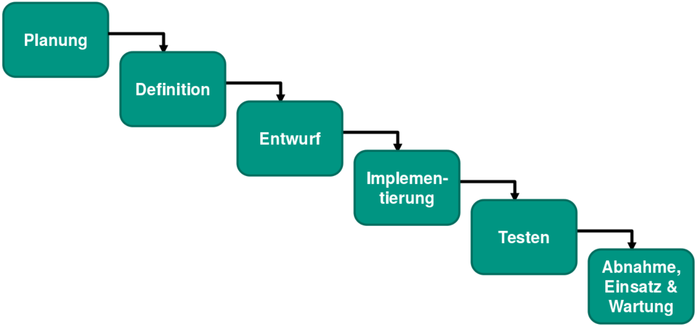
STEINBUCH CENTER FOR COMPUTING



About PSE

Praxis der Softwareentwicklung(PSE) = Software Engineering Practice

- Waterfall model
  - Planning/definition



Planung

Definition

Entwurf

Implementierung

Testen

Abnahme, Einsatz & Wartung

Purpose

Node.js bindings for ROOT

- Be able to write ROOT code in Node.js programs
- Integrate ROOT into Node.js based web applications

Required Criteria

The bindings must

- Work on Linux
- Allow the user to interact with any ROOT class from the Node.js JavaScript interpreter
- Accept C++ code for just-in-time compilation
- Update dynamically following changes to C++ internals
- Provide asynchronous wrappers for common I/O operations (i.e. file and tree access)

Optional Criteria

The bindings should

- 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)

Limiting criteria

The bindings should not

- Add any extending functionality to the existing ROOT framework
- Necessarily support previous/future ROOT versions

Product usage

- It’s JavaScript → web-applications
- Expose processed data and then visualize it locally
- Interact with remote data (i.e. streamed via RPC)
- Accessible on ’unconvential’ devices (mobile phones/tablets)

Audience

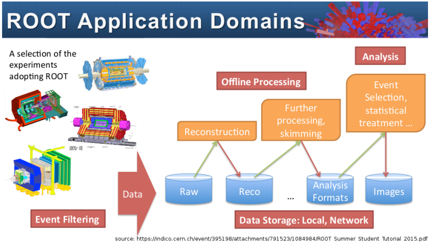
- Scientists (e.g. particle physicists) and Researchers
- Typical user will know ROOT and JavaScript → rather technology proficient
- Web-developers

Operating conditions

- Servers that run ROOT
- ROOT6 is currently only available on Mac and Linux, so that's our focus

ROOT

- Process and visualize large amounts of scientific data (CERN)
- Features a C++ interpreter (CLING) - i.e. used for rapid and efficient prototyping
- Persistency mechanism for C++ objects



ROOT Application Domains

A selection of the experiments adopting ROOT

Analysis

Offline Processing

Event Selection, statistical treatment ...

Further processing, skimming

Reconstruction

Data

Raw

Reco

...

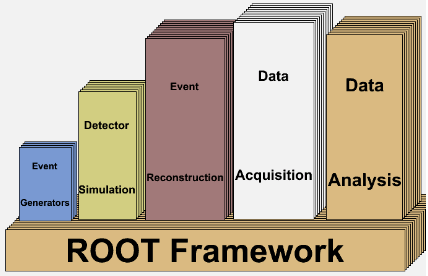
Analysis Formats

Images

Event Filtering

Data Storage: Local, Network

source: https://indico.cern.ch/event/395198/attachments/791523/1084984/ROOT Summer Student Tutorial 2015.pdf



Event Generators

Detector Simulation

Event Reconstruction

Data Acquisition

Data Analysis

ROOT Framework

Node.js

- Open source runtime environment
  - Develop server side web applications
  - Act as a stand alone web server

Node.js

- Open source runtime environment
  - Develop server side web applications
  - Act as a stand alone web server
- Google V8 engine to execute JavaScript code

Node.js

- Open source runtime environment
  - Develop server side web applications
  - Act as a stand alone web server
- Google V8 engine to execute JavaScript code
- rootJS bindings realized as native Node.js module written in C++

Hardware

- Task: encapsulation of ROOT objects and functions
  - → Scanning ROOT structures during initialization
  - → Encapsulating objects with heavily nested object structures
  - → Introduce (proxy) object cache

Hardware

- Task: encapsulation of ROOT objects and functions
  - → Scanning ROOT structures during initialization
  - → Encapsulating objects with heavily nested object structures
  - → Introduce (proxy) object cache

⇒ Generally negligible hardware requirements of the bindings themselves

Product data

The following data will be stored by the rootJS bindings

- All ROOT classes and methods as they dynamically mapped to their JavaScript equivalents
- ROOT environment state
- Application context is derived from TApplication
- Map of v8::handles 2 identified by the address of ROOT objects

Scenarios

rootJS is used by applications to access the ROOT framework

Scenarios

rootJS is used by applications to access the ROOT framework
⇒ our users are those applications

Event Viewer

Event Viewer

Event Viewers provide visualisation of experimental data

Event Viewer

Event Viewers provide visualisation of experimental data

- useful for quick eyescan of data

Event Viewer

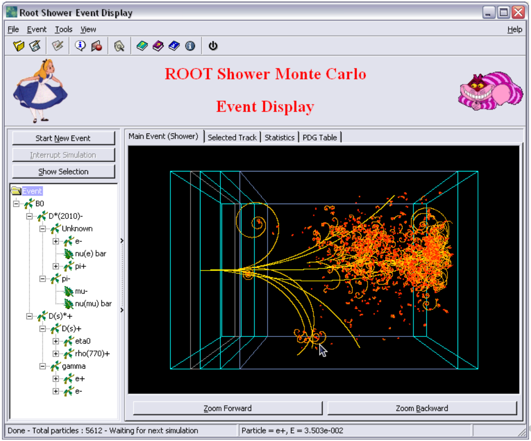
Event Viewers provide visualisation of experimental data

- useful for quick eyescan of data
- check if data is recorded properly

Event Viewer

Event Viewers provide visualisation of experimental data

- useful for quick eyescan of data
- check if data is recorded properly



Done - Total particles : 5612 - Waiting for next simulation

Particle = e+, E = 3.503e-002

Event Viewer

Conventional ROOT Event Viewer

Event Viewer

Conventional ROOT Event Viewer

- standalone ROOT application

Event Viewer

Conventional ROOT Event Viewer

- standalone ROOT application
- requires ROOT on machine

Event Viewer

Conventional ROOT Event Viewer

- standalone ROOT application
- requires ROOT on machine
- requires ROOT's dependencies

Event Viewer

Conventional ROOT Event Viewer

- standalone ROOT application
- requires ROOT on machine
- requires ROOT's dependencies
- requires access to data source

Event Viewer

Conventional ROOT Event Viewer

- standalone ROOT application
- requires ROOT on machine
- requires ROOT's dependencies
- requires access to data source

⇒ very limited portability and harsh requirements for client system

Event Viewer

Client/Server based Web Event Viewer using rootJS and nodeJS

Event Viewer

Client/Server based Web Event Viewer using rootJS and nodeJS

- server runs ROOT and its dependencies

Event Viewer

Client/Server based Web Event Viewer using rootJS and nodeJS

- server runs ROOT and its dependencies
- no access to critical data sources required

Event Viewer

Client/Server based Web Event Viewer using rootJS and nodeJS

- server runs ROOT and its dependencies
- no access to critical data sources required
- no heavy work load on client system

Event Viewer

Client/Server based Web Event Viewer using rootJS and nodeJS

- server runs ROOT and its dependencies
- no access to critical data sources required
- no heavy work load on client system
- client only needs modern web browser

Event Viewer

Client/Server based Web Event Viewer using rootJS and nodeJS

- server runs ROOT and its dependencies
- no access to critical data sources required
- no heavy work load on client system
- client only needs modern web browser

⇒ great portability and ease of use as client can be almost any device

Event Viewer

Scenario name	`EventViewer`
Participating actors	Server:EventViewerServer; :ROOT Client:EventViewerClient; :rootJS
Flow of events	■ `Client` requests updates from `Server`. ■ `Server` interfaces with `ROOT` through `rootJS`. ■ `ROOT`’s I/O accesses and processes data ■ `ROOT` returns the data to the `Server` using `rootJS`. ■ `Server` sends the data to the `Client` ■ `Client` renders data locally.

Scenarios

In what ways could these bindings also improve work efficiency for scientists?

Scenarios

In what ways could these bindings also improve work efficiency for scientists?

- integrating run logs and quality assurance in ROOT workflow

Scenarios

In what ways could these bindings also improve work efficiency for scientists?

- integrating run logs and quality assurance in ROOT workflow
- ...

UseROOTGlobal

Use case name	`UseROOTGlobal`
Participating actor instances	Initiated by `NodeJSApplication`; Processed by `rootJS`; Communicates with `ROOT`
Flow of events	1. The `NodeJSApplication` requests access to a global variable of `ROOT`. 2. `rootJS` sends a request to the corresponding `ROOT` variable. 3. `ROOT` returns the requested variable value. 4. The value is passed from `rootJS` to the `NodeJSApplication`.

UseROOTGlobal

Entry condition	`rootJS` has been initialized.
Exit condition	The value has been returned to the `NodeJSApplication`.

UseROOTObject

Use case name	`UseROOTObject`
Participating actor instances	Initiated by `NodeJSApplication`; Processed by `rootJS`, `ProxyObject`; Communicates with `ROOT`
Flow of events	1. The `NodeJSApplication` requests access to a `ROOT` object by calling a constructor function. 2. `rootJS` encapsulates the requested `ROOT` object within a `ProxyObject` that was created recursively.

UseROOTObject

Flow of events

3. rootJS stores the created ProxyObject in a cache memory.
4. The ProxyObject is exposed to the NodeJSApplication.

Entry condition	rootJS has been initialized.
Exit condition	The reference of the ProxyObject has been return to the NodeJSApplication.

UseROOTFunction

Use case name	`UseROOTFunction`
Participating actor instances	Initiated by `NodeJSApplication`; Processed by `rootJS`, `ProxyObject`; Communicates with `ROOT`
Flow of events	1. The `NodeJSApplication` requests access to a `ROOT` function. 2. `rootJS` calls the corresponding `ROOT` function. 3. `ROOT` responds.

UseROOTFunction

Flow of events

4. `rootJS` encapsulates the returned `ROOT` object within a `ProxyObject`.
5. The `ProxyObject` is exposed to the `NodeJSApplication`.

Entry condition	`rootJS` has been initialized.
Exit condition	The reference of the `ProxyObject` has been return to the `NodeJSApplication`.

UseJIT

Use Case name	UseJIT
Participating actor instances	Initiated by NodeJSApplication; Processed by rootJS, Cling; Communicates with ROOT
Flow of events	1. The NodeJSApplication wants to execute ROOT specific C++ code (given as string) during runtime. 2. rootJS forwards the instructions to Cling. 3. Cling evaluates the received instructions using JIT compilation concepts and dynamically modifies the state of ROOT.

UseJIT

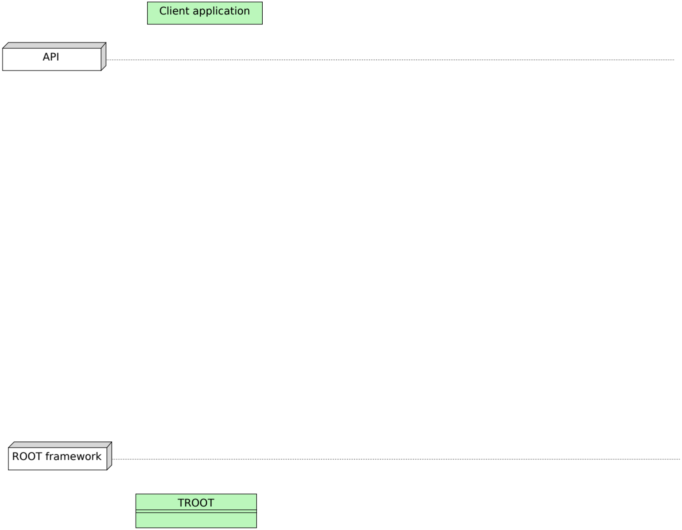
Flow of events

3. `rootJS` takes care of encapsulating exceptions possibly thrown by `Cling` or `ROOT` during evaluation and execution.
4. `rootJS` provides the evaluation results and corresponding return values to the `NodeJSApplication`.

Entry condition	`rootJS` and `Cling` have been initialized.
Exit condition	`rootJS` either confirms the proper execution of the specified instructions or forwards thrown exceptions to the `NodeJSApplication`.

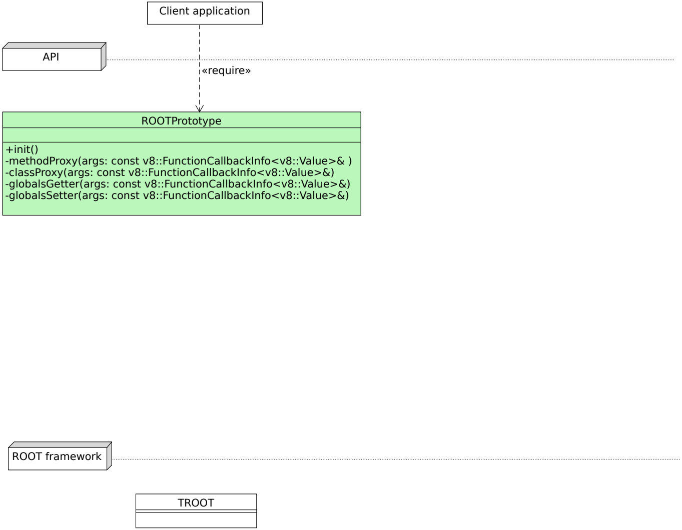
Basic Architecture

→



Basic Architecture

→



Client application

API

«require»

ROOTPrototype

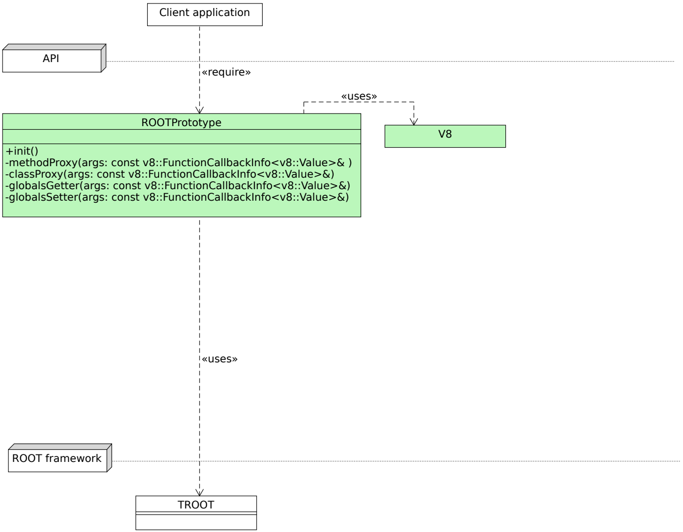
+init()
-methodProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-classProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsGetter(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsSetter(args: const v8::FunctionCallbackInfo<v8::Value>&)

ROOT framework

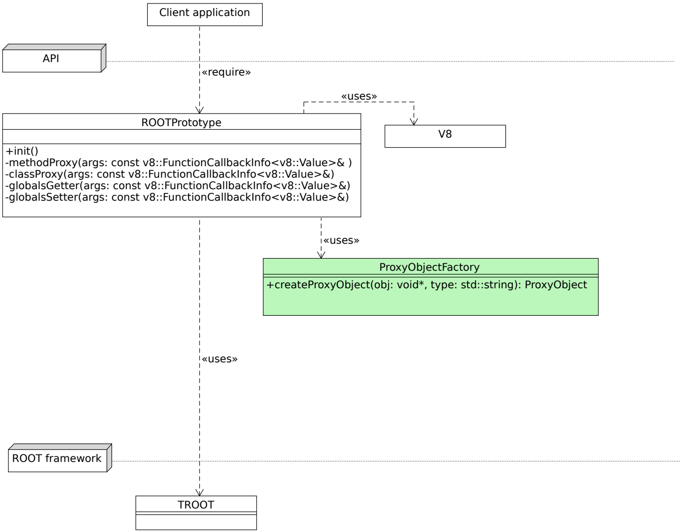
TROOT

Basic Architecture

→


Client application
API
«require»
«uses»
ROOTPrototype
+init()
-methodProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-classProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsGetter(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsSetter(args: const v8::FunctionCallbackInfo<v8::Value>&)
V8
«uses»
ROOT framework
TROOT

Basic Architecture



Client application

API

«require»

«uses»

ROOTPrototype

+init()
-methodProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-classProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsGetter(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsSetter(args: const v8::FunctionCallbackInfo<v8::Value>&)

V8

«uses»

→

ProxyObjectFactory

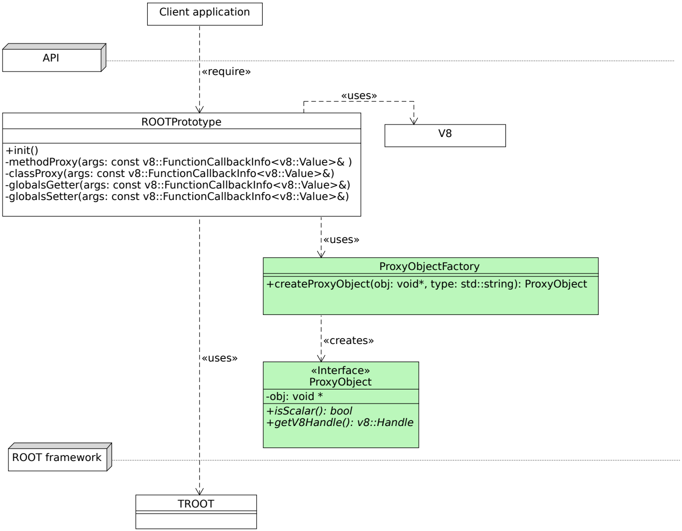
+createProxyObject(obj: void*, type: std::string): ProxyObject

«uses»

ROOT framework

TROOT

Basic Architecture



Client application

API

«require»

«uses»

ROOTPrototype

+init()
-methodProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-classProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsGetter(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsSetter(args: const v8::FunctionCallbackInfo<v8::Value>&)

V8

«uses»

ProxyObjectFactory

+createProxyObject(obj: void*, type: std::string): ProxyObject

«creates»

«uses»

«Interface»
ProxyObject

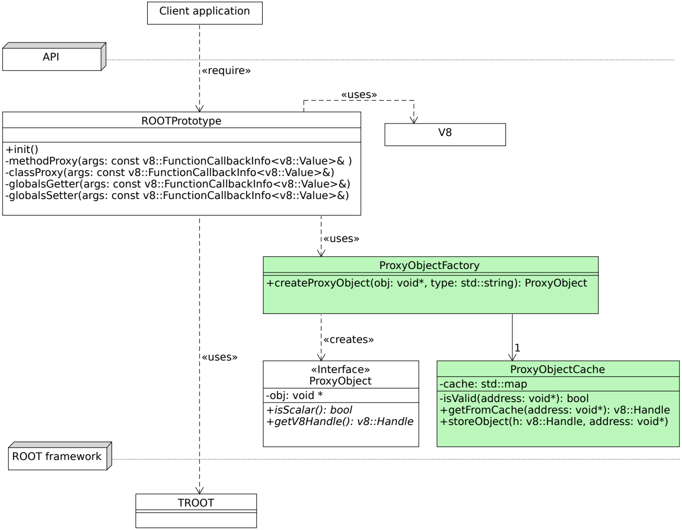
-obj: void *

+isScalar(): bool
+getV8Handle(): v8::Handle

ROOT framework

TROOT

Basic Architecture


Client application

API

«require»

«uses»

ROOTPrototype

+init()
-methodProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-classProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsGetter(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsSetter(args: const v8::FunctionCallbackInfo<v8::Value>&)

V8

«uses»

ProxyObjectFactory

+createProxyObject(obj: void*, type: std::string): ProxyObject

«creates»

1

«uses»

«Interface»
ProxyObject

-obj: void *

+isScalar(): bool
+getV8Handle(): v8::Handle

ProxyObjectCache

-cache: std::map

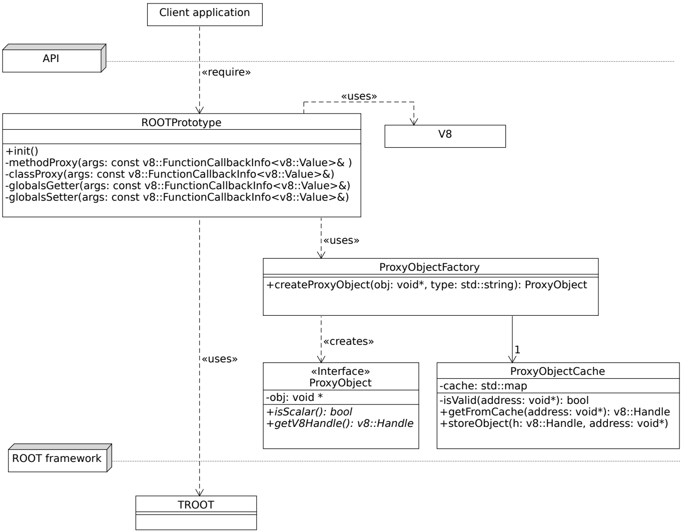
-isValid(address: void*): bool
+getFromCache(address: void*): v8::Handle
+storeObject(h: v8::Handle, address: void*)

ROOT framework

TROOT

→

Basic Architecture



Client application

API

«require»

«uses»

ROOTPrototype

+init()
-methodProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-classProxy(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsGetter(args: const v8::FunctionCallbackInfo<v8::Value>&)
-globalsSetter(args: const v8::FunctionCallbackInfo<v8::Value>&)

V8

«uses»

ProxyObjectFactory

+createProxyObject(obj: void*, type: std::string): ProxyObject

«creates»

1

«uses»

«Interface»
ProxyObject

-obj: void *

+isScalar(): bool
+getV8Handle(): v8::Handle

ProxyObjectCache

-cache: std::map

-isValid(address: void*): bool
+getFromCache(address: void*): v8::Handle
+storeObject(h: v8::Handle, address: void*)

ROOT framework

TROOT

Initialization

- Expose all
  - Global variables
  - Global functions
  - Classes

Initialization

- Expose all
  - Global variables
  - Global functions
  - Classes
- Each are bound to corresponding proxy methods
- An object which members are the exposed features is beeing passed to node

Initialization

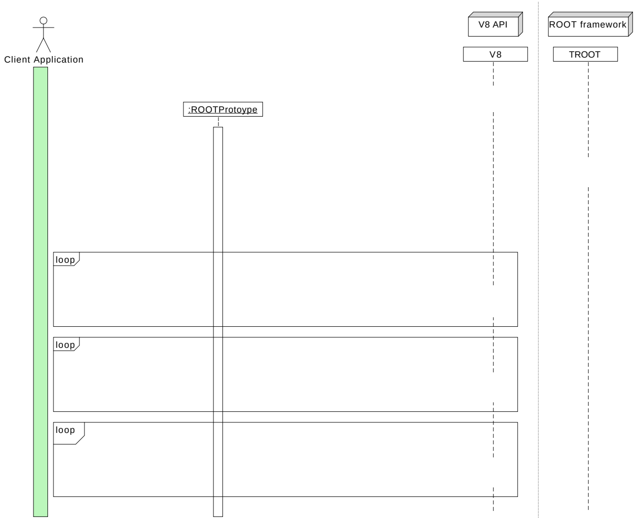
- Expose all
  - Global variables
  - Global functions
  - Classes
- Each are bound to corresponding proxy methods
- An object which members are the exposed features is beeing passed to node

Names

- Functions and classes have the same name as in Root
- Global variables can be called using Get[Variable] and Set[Variable] methods

rootJS init

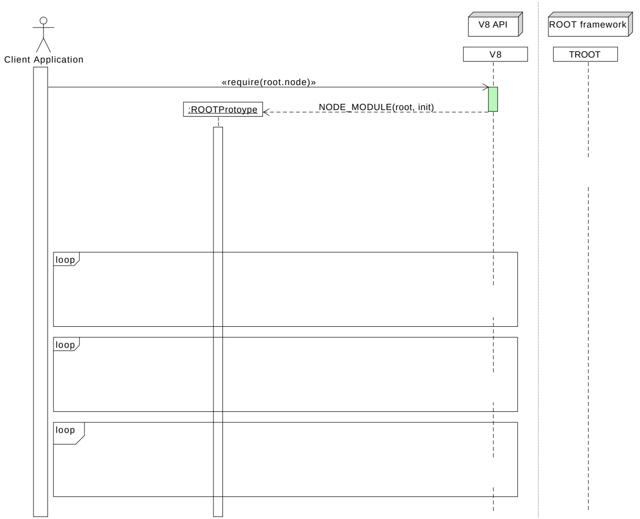
→



Client Application
V8 API
V8
ROOT framework
TROOT
:ROOTProtoype
loop
loop
loop

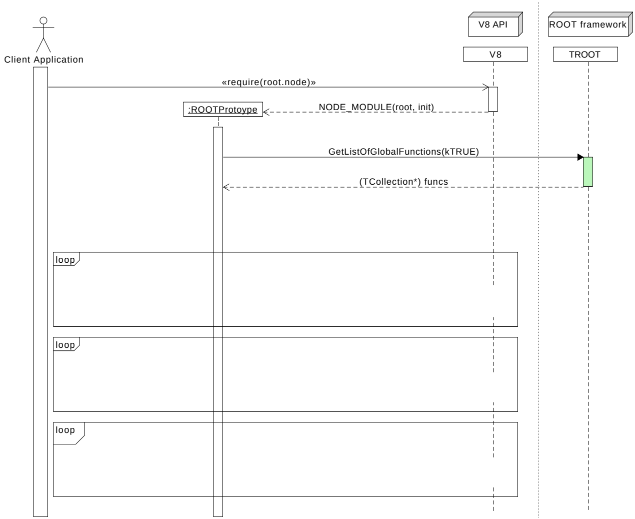
rootJS init

→


Client Application
V8 API
ROOT framework
V8
TROOT
«require(root.node)»
:ROOTProtoype
NODE_MODULE(root, init)
loop
loop
loop

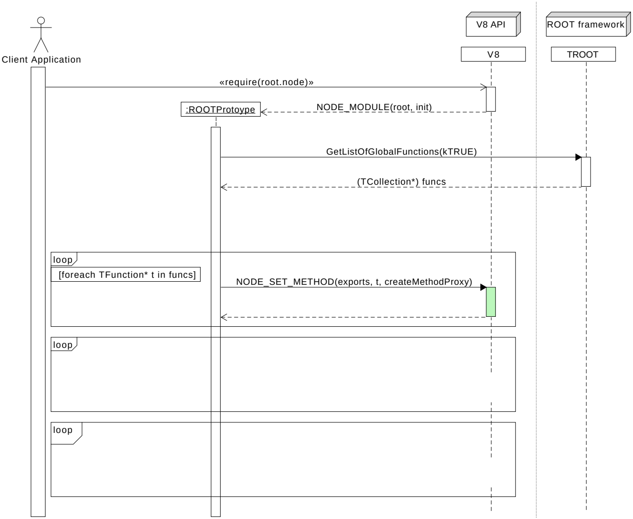
rootJS init

→



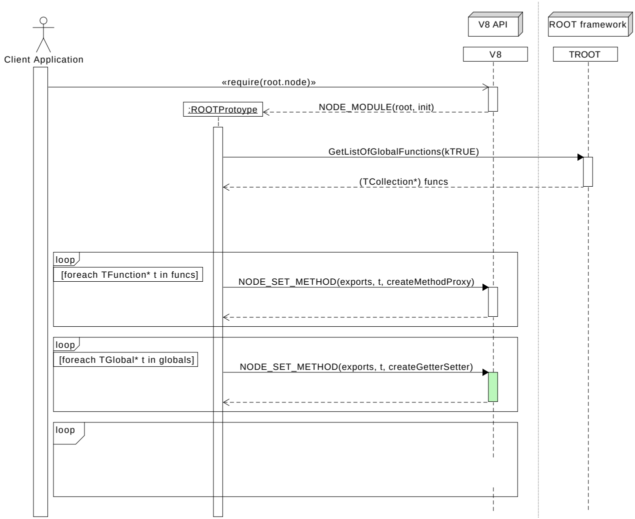
V8 API
ROOT framework
Client Application
V8
TROOT
«require(root.node)»
:ROOTProtoype
NODE_MODULE(root, init)
GetListOfGlobalFunctions(kTRUE)
(TCollection*) funcs
loop
loop
loop

rootJS init


V8 API
ROOT framework
Client Application
V8
TROOT
«require(root.node)»
:ROOTProtoype
NODE_MODULE(root, init)
GetListOfGlobalFunctions(kTRUE)
(TCollection*) funcs
loop
[foreach TFunction* t in funcs]
NODE_SET_METHOD(exports, t, createMethodProxy)
loop
loop

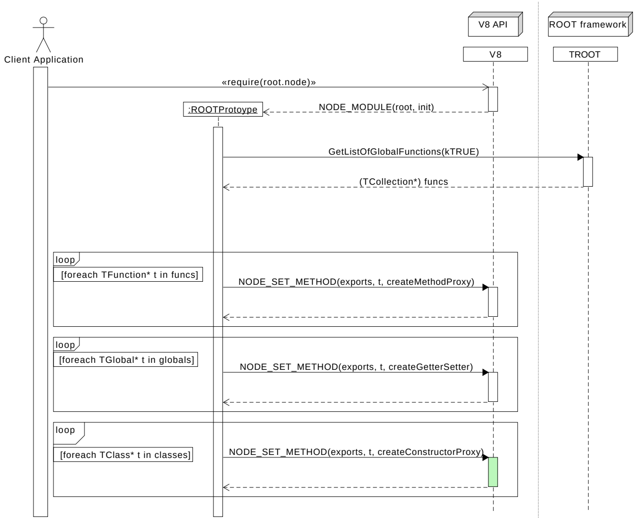
→

rootJS init


V8 API
ROOT framework
Client Application
V8
TROOT
«require(root.node)»
:ROOTProtoype
NODE_MODULE(root, init)
GetListOfGlobalFunctions(kTRUE)
(TCollection*) funcs
loop
[foreach TFunction* t in funcs]
NODE_SET_METHOD(exports, t, createMethodProxy)
loop
[foreach TGlobal* t in globals]
NODE_SET_METHOD(exports, t, createGetterSetter)
loop

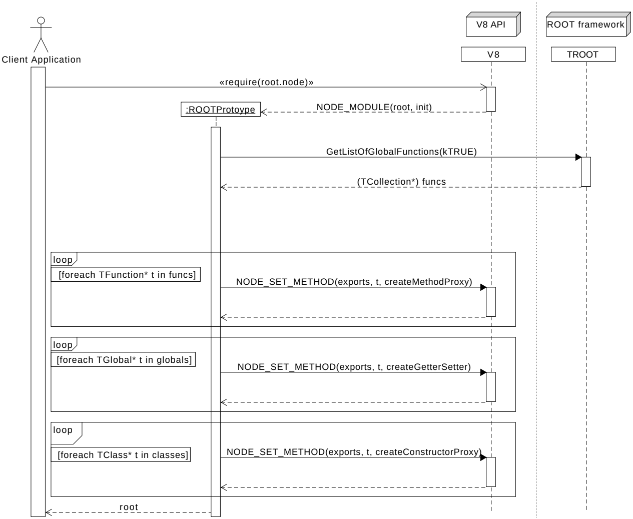
→

rootJS init


V8 API
ROOT framework
Client Application
V8
TROOT
«require(root.node)»
:ROOTProtoype
NODE_MODULE(root, init)
GetListOfGlobalFunctions(kTRUE)
(TCollection*) funcs
loop
[foreach TFunction* t in funcs]
NODE_SET_METHOD(exports, t, createMethodProxy)
loop
[foreach TGlobal* t in globals]
NODE_SET_METHOD(exports, t, createGetterSetter)
loop
[foreach TClass* t in classes]
NODE_SET_METHOD(exports, t, createConstructorProxy)

→

rootJS init



Client Application
V8 API
ROOT framework
V8
TROOT
«require(root.node)»
:ROOTProtoype
NODE_MODULE(root, init)
GetListOfGlobalFunctions(kTRUE)
(TCollection*) funcs
loop
[foreach TFunction* t in funcs]
NODE_SET_METHOD(exports, t, createMethodProxy)
loop
[foreach TGlobal* t in globals]
NODE_SET_METHOD(exports, t, createGetterSetter)
loop
[foreach TClass* t in classes]
NODE_SET_METHOD(exports, t, createConstructorProxy)
root

Call a feature

- All features in node are mapped to a proxy method that will be called

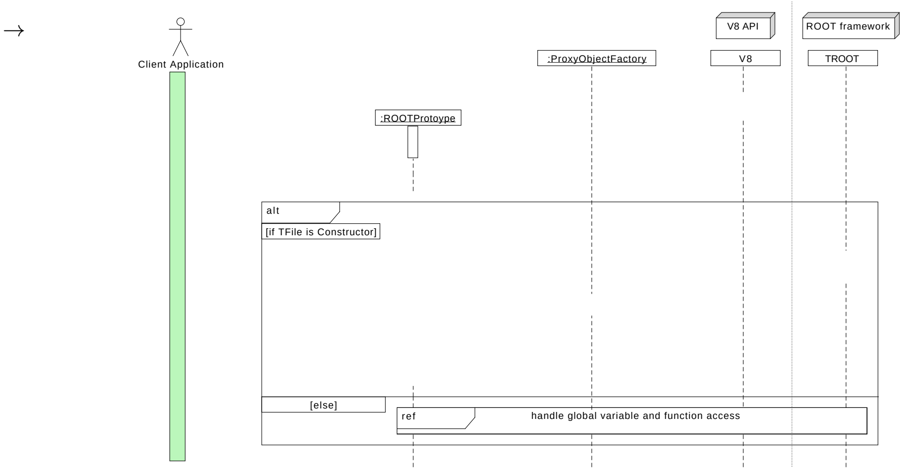
Call a feature

- All features in node are mapped to a proxy method that will be called
- The proxy method will eventually call a root function and pass the result to our ObjectFactory

Call a feature

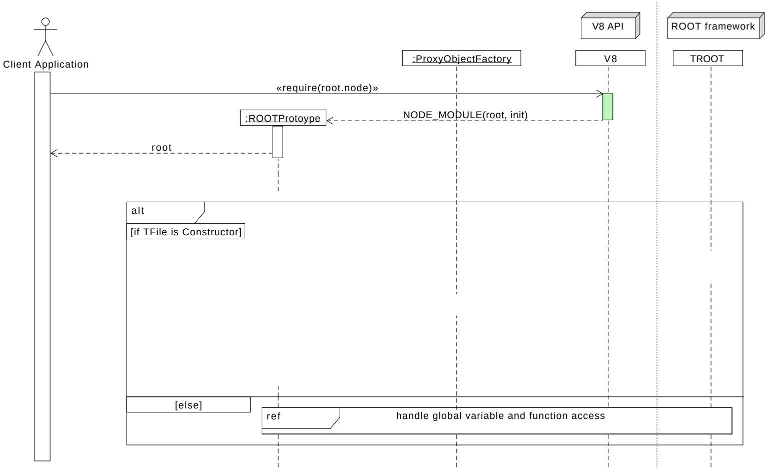
- All features in node are mapped to a proxy method that will be called
- The proxy method will eventually call a root function and pass the result to our ObjectFactory
- By looking at the object type an corresponding v8::Handle will be generated and returned to node
  - If the result is an object this will be done recursively

proxied file access

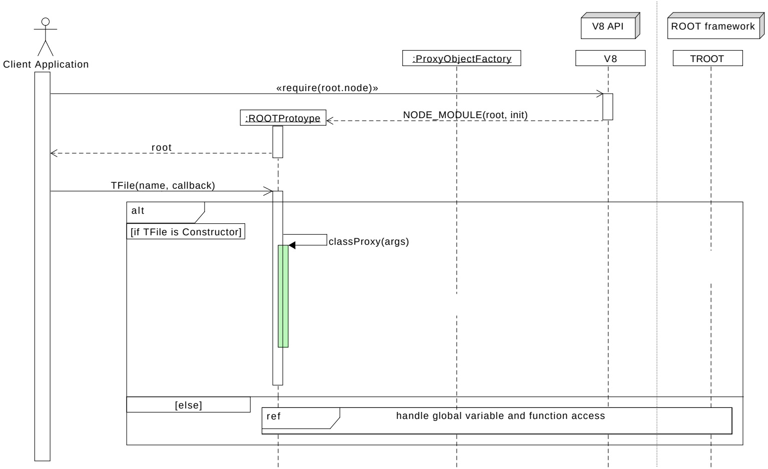


proxied file access

→



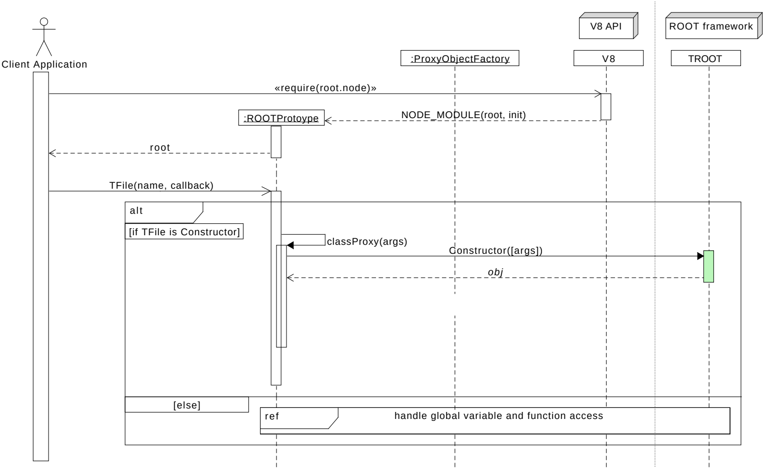
proxied file access



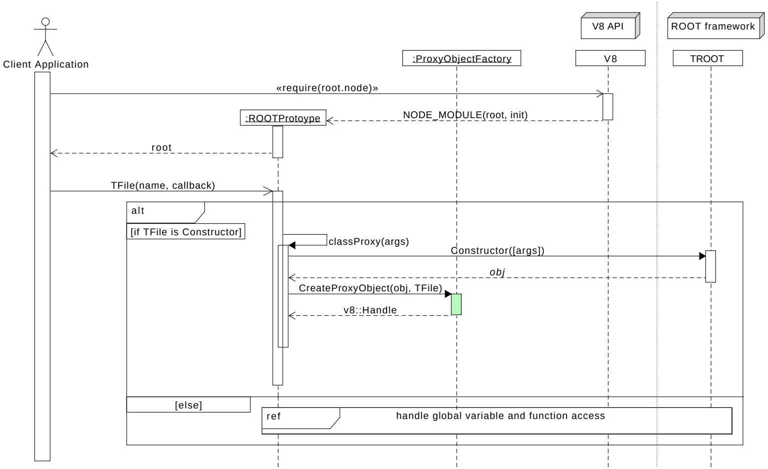
→

proxied file access

→

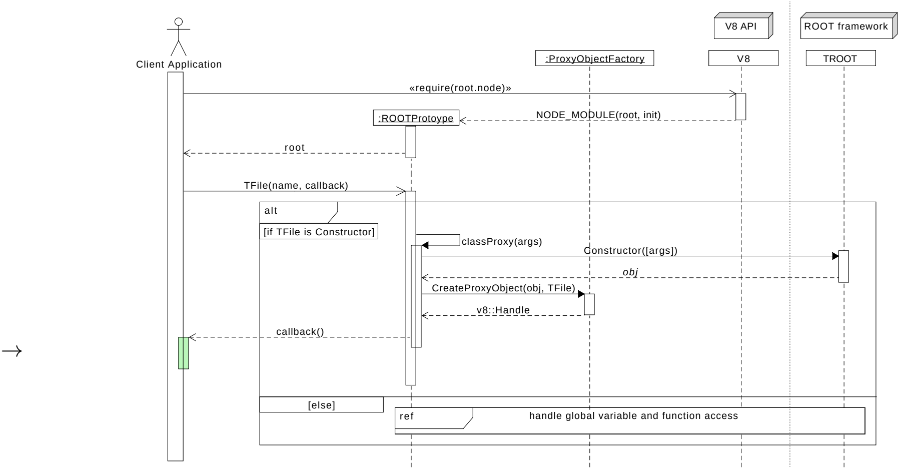


proxied file access

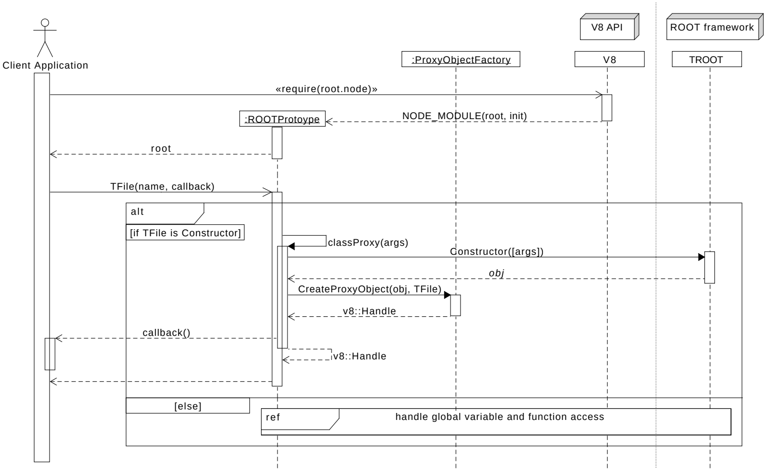


→

proxied file access



proxied file access



Test Cases

- Make sure all elements are callable without crashing
- To verify results of function calls and calculations, we would need to run ROOT's testcases
- Porting all ROOT testcases would make no sense!
- Only port a subset to make sure the bindings are working and leave the rest to the ROOT developers

References I