rootJS – Node.js bindings for ROOT 6
PSE – Software Engineering Practice

J. Schwabe, C. Haas, T. Beffart, M. Früeh, S. Rajgopal, C. Wolff
Introduction – the team

- Computer Science students 3rd semester
- Supervisor: Dr. Szuba
- Team members
  - Christoph Haas
  - Jonas Schwabe
  - Theo Beffart
  - Maximilian Früh
  - Christoph Wolff
  - Sachin Rajgopal
Introduction – 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**

Introduction – 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
Introduction - 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++
Introduction - rootJS

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

- System Requirements
  - Mac OS X and Linux
  - ROOT 6
  - Node.js versions
    - Stable on Node.js 4.4 (LTS)
Introduction – What is PSE?

- **Praxis der Softwareentwicklung (PSE)**
- Create software in a team in 5 months using object oriented software engineering
- Design: UML
- The final software: Maximum of 10k LOC, 250 hours/person
- Weekly meetings
- Development phases - waterfall model

```
Analysis

Requirement Elicitation

System Design

Implementation

Testing & Integration

Operation & Maintenance
```
Phase Recap – Requirement Elicitation

- Required criteria
Phase Recap – Requirement Elicitation

- Required criteria
  - Work on Linux
Phase Recap – Requirement Elicitation

- Required criteria
  - Work on Linux
  - Accept C++ code for JIT compilation
Phase Recap – Requirement Elicitation

- Required criteria
  - Work on Linux
  - Accept C++ code for JIT compilation
  - Dynamically update C++ internals on changes
Phase Recap – Requirement Elicitation

- Required criteria
  - Work on Linux
  - Accept C++ code for JIT compilation
  - Dynamically update C++ internals on changes
  - Asynchronous wrappers for common I/O operations
Phase Recap – Requirement Elicitation

- Required criteria
  - Work on Linux
  - Accept C++ code for JIT compilation
  - Dynamically update C++ internals on changes
  - Asynchronous wrappers for common I/O operations

- Limiting criteria
Phase Recap – Requirement Elicitation

- **Required criteria**
  - Work on Linux
  - Accept C++ code for JIT compilation
  - Dynamically update C++ internals on changes
  - Asynchronous wrappers for common I/O operations

- **Limiting criteria**
  - Do not extend existing ROOT functionality
Phase Recap – Requirement Elicitation

- **Required criteria**
  - Work on Linux
  - Accept C++ code for JIT compilation
  - Dynamically update C++ internals on changes
  - Asynchronous wrappers for common I/O operations

- **Limiting criteria**
  - Do not extend existing ROOT functionality
  - Do not necessarily support future ROOT versions
Phase Recap – Requirement Elicitation

- Language bindings
Phase Recap – Requirement Elicitation

- Language bindings
  - Use ROOT functions
Phase Recap – Requirement Elicitation

- Language bindings
  - Use ROOT functions
  - Use ROOT objects
Phase Recap – Requirement Elicitation

- Language bindings
  - Use ROOT functions
  - Use ROOT objects
  - Use JIT compiler
Phase Recap – Requirement Elicitation

- Language bindings
  - Use ROOT functions
  - Use ROOT objects
  - Use JIT compiler
- Focus on benefits provided by JavaScript
Phase Recap – Requirement Elicitation

- Language bindings
  - Use ROOT functions
  - Use ROOT objects
  - Use JIT compiler

- Focus on benefits provided by JavaScript
  - Asynchronous calls
Phase Recap – Requirement Elicitation

- Language bindings
  - Use ROOT functions
  - Use ROOT objects
  - Use JIT compiler

- Focus on benefits provided by JavaScript
  - Asynchronous calls
  - Use in web applications
Phase Recap – Requirement Elicitation

- Usage scenario: event viewer
Phase Recap – Requirement Elicitation

- Usage scenario: event viewer
  - Visualizes experimental data
Phase Recap – Requirement Elicitation

- Usage scenario: event viewer
- Visualizes experimental data
Phase Recap – Requirement Elicitation

- Usage scenario: event viewer
  - Visualizes experimental data
  - Standalone ROOT application
Phase Recap – Requirement Elicitation

- Usage scenario: event viewer
  - Visualizes experimental data
  - Standalone ROOT application
  - Needs ROOT and dependencies installed
Phase Recap – Requirement Elicitation

- Usage scenario: event viewer
  - Visualizes experimental data
  - Standalone ROOT application
  - Needs ROOT and dependencies installed
  - Needs access to data sources
Phase Recap – Requirement Elicitation

- Usage scenario: event viewer
  - Visualizes experimental data
  - Standalone ROOT application
  - Needs ROOT and dependencies installed
  - Needs access to data sources
  - Limited portability
Phase Recap – Requirement Elicitation

- Client / Server approach using rootJS
Phase Recap – Requirement Elicitation

- Client / Server approach using rootJS
  - Server runs ROOT and dependencies, rootJS

ROOT + dependencies, rootJS
Phase Recap – Requirement Elicitation

- Client / Server approach using rootJS
  - Server runs ROOT and dependencies, rootJS

Data sources

ROOT + dependencies, rootJS
Phase Recap – Requirement Elicitation

- Client / Server approach using rootJS
  - Server runs ROOT and dependencies, rootJS
  - Client only needs modern web browser
Phase Recap – Requirement Elicitation

- Client / Server approach using rootJS
  - Server runs ROOT and dependencies, rootJS
  - Client only needs modern web browser
  - No heavy work load on client
Phase Recap – Design
Phase Recap – Design

- Basic architecture requirements:
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks
- fundamental language differences
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks
- fundamental language differences
  - different type systems (dynamic vs. static)
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks
- fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks

- fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks

- fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?

- Task: „write an adapter“
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks

- fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?

- Task: „write an adapter“
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks

- fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?

- Task: „write an adapter“
  - software design pattern
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks
- Fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?
- Task: „write an adapter“
  - software design pattern
  - help incompatible interfaces to work together
Phase Recap – Design

Basic architecture requirements:
- dynamic object creation and encapsulation
- non-blocking function calls via callbacks

Fundamental language differences
- different type systems (dynamic vs. static)
- prototype functions instead of classes
- multithreading support?

Task: „write an adapter“
- software design pattern
- help incompatible interfaces to work together

Environment:
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks

- fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?

- Task: “write an adapter“
  - software design pattern
  - help incompatible interfaces to work together

- Environment:
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks

- Fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?

- Task: „write an adapter“
  - software design pattern
  - help incompatible interfaces to work together

- Environment:
  - v8 API:
    - object exposure and callback handling
Phase Recap – Design

- Basic architecture requirements:
  - dynamic object creation and encapsulation
  - non-blocking function calls via callbacks

- Fundamental language differences
  - different type systems (dynamic vs. static)
  - prototype functions instead of classes
  - multithreading support?

- Task: “write an adapter“
  - software design pattern
  - help incompatible interfaces to work together

- Environment:
  - v8 API:
    - object exposure and callback handling
  - ROOT RTTI-interface
    - class, namespace, global and member variable information
Design – Requirements Realization
Design – Requirements Realization

init
Design – Requirements Realization

- init

  recursively seek & expose
classes and namespaces
Design – Requirements Realization

- recursively seek & expose classes and namespaces
Design – Requirements Realization

- init
- callback handling

- recursively seek & expose classes and namespaces

- entry point for client interactions with ROOT
Design – Requirements Realization

- init
- callback handling
- entry point for client interactions with ROOT
- recursively seek & expose classes and namespaces

- function calls
- object access
- object creation
Design – Requirements Realization

- init
  - recursively seek & expose classes and namespaces
- callback handling
- object access
- object creation
- entry point for client interactions with ROOT

- provide async call context before forwarding to RTTI API
Design – Requirements Realization

- init
  - recursively seek & expose classes and namespaces
  - provide async call context before forwarding to RTTI API

- callback handling
- function calls
- object access
  - direct access to C++ objects in memory via corresponding proxy object

- object creation
  - entry point for client interactions with ROOT
Design – Requirements Realization

- init
  - recursively seek & expose classes and namespaces

- function calls
  - provide async call context before forwarding to RTTI API

- object access
  - direct access to C++ objects in memory via corresponding proxy object

- object creation
  - forward constructor calls
  - encapsulate constructed objects for exposure

- callback handling
  - entry point for client interactions with ROOT
Design – Requirements Realization

- init
  - recursively seek & expose classes and namespaces

- function calls
  - provide async call context before forwarding to RTTI API

- callback handling
  - entry point for client interactions with ROOT

- object access
  - direct access to C++ objects in memory via corresponding proxy object

- JavaScript object
  - Proxy object
    - C++ object
      - object creation
        - forward constructor calls
        - encapsulate constructed objects for exposure
Design – Architecture Concept

- init
- callback handling
- function calls
- object access
- object creation
Design – Architecture Concept

- init
- callback handling
- function calls
- object access
- object creation

FunctionProxyFactory
+ fromArgs()
Design – Architecture Concept

- init
  - function calls
    - FunctionProxyFactory
      - + fromArgs()
    - FunctionProxy
      - + call()
  - callback handling
    - object access
    - object creation
Design – Architecture Concept

- init
- callback handling

- function calls
- object access
- object creation

FunctionProxyFactory
  + fromArgs()

FunctionProxy
  + call()
Design – Architecture Concept

- **init**
- **callback handling**
- **function calls**
- **object access**
- **object creation**

**FunctionProxyFactory**
- **FunctionProxy**
  - + call()
  - + fromArgs()

**ObjectProxy**
- + readValue()
- + writeValue()
Design – Architecture Concept

- **init**
- **callback handling**
- **function calls**
  - `FunctionProxyFactory`
    - `+ fromArgs()`
  - `FunctionProxy`
    - `+ call()`
- **object access**
  - `ObjectProxy`
    - `+ readValue()`
    - `+ writeValue()`
- **object creation**
  - `ObjectProxyFactory`
    - `+ createCapsule()`
Design – Architecture Concept

init

callback handling

function calls

object access

object creation

FunctionProxyFactory
  + fromArgs()

ObjectProxy
  + readValue()
  + writeValue()

ObjectProxyFactory
  + createCapsule()

TemplateFactory
  + createTmplt()
Design – Architecture Concept

FunctionProxyFactory
  + fromArgs()

FunctionProxy
  + call()

ObjectProxy
  + readValue()
  + writeValue()

ObjectProxyFactory
  + createCapsule()

ObjectProxy
  - address

ObjectProxyFactory
  + createTmplt()

TemplateFactory

callback handling

init

function calls

object access

object creation
Design – Architecture Concept

- **init**
  - callback handling
    - function calls
      - `FunctionProxyFactory`
        - + `fromArgs()`
          - `FunctionProxy`
            - + `call()`
    - object access
      - `ObjectProxy`
        - + `readValue()`
          - `ObjectProxyFactory`
            - + `createCapsule()`
    - object creation
      - `ObjectProxy`
        - + writeValue()
      - `ObjectProxyFactory`
        - + createTmplt()
      - `TemplateFactory`
        - + createTmplt()
Design – Architecture Concept

FunctionProxyFactory
  + fromArgs()

FunctionProxy
  + call()

ObjectProxy
  + readValue()
  + writeValue()

ObjectProxyFactory
  + createCapsule()

ObjectProxy
  - address

TemplateFactory
  + createTmplt()
Design – Core Architecture
Design – Core Architecture

NodeHandler

+ exposeROOT()
Design – Core Architecture

NodeHandler
+ exposeROOT()

CallbackHandler
+ onAccess()
Design – Core Architecture

NodeHandler
+ exposeROOT()

CallbackHandler
+ onAccess()

FunctionProxyFactory
+ fromArgs()
Design – Core Architecture

NodeHandler
  + exposeROOT()

CallbackHandler
  + onAccess()

FunctionProxyFactory
  + fromArgs()

FunctionProxy
  + call()
Design – Core Architecture

- **NodeHandler**
  - `+ exposeROOT()`

- **CallbackHandler**
  - `+ onAccess()`

- **FunctionProxyFactory**
  - `+ fromArgs()`

- **ObjectProxyFactory**
  - `+ createCapsule()`

- **FunctionProxy**
  - `+ call()`
Design – Core Architecture

- **NodeHandler**
  - `+ exposeROOT()`

- **CallbackHandler**
  - `+ onAccess()`

- **FunctionProxyFactory**
  - `+ fromArgs()`

- **ObjectProxyFactory**
  - `+ createCapsule()`

- **FunctionProxy**
  - `+ call()`

- **ObjectProxy**
  - `+ read() / write()`
Design – Core Architecture

- **NodeHandler**
  - `+ exposeROOT()`

- **CallbackHandler**
  - `+ onAccess()`

- **FunctionProxyFactory**
  - `+ fromArgs()`

- **ObjectProxyFactory**
  - `+ createCapsule()`

- **Proxy**
  - `- address`

- **FunctionProxy**
  - `+ call()`

- **ObjectProxy**
  - `+ read() / write()`
Design – Core Architecture

NodeHandler
  + exposeROOT()

CallbackHandler
  + onAccess()

FunctionProxyFactory
  + fromArgs()

ObjectProxyFactory
  + createCapsule()

Proxy
  - address

FunctionProxy
  + call()

ObjectProxy
  + read() / write()

PrimitiveProxy
Implementation – Principles
Implementation – Principles

- Test driven development
  - Tests for features
  - Test for encountered bugs
  - Tests rely on ROOT behaviour
Implementation – Principles

- Test driven development
  - Tests for features
  - Test for encountered bugs
  - Tests rely on ROOT behaviour

- Stable master branch
  - Features / bug fixes on separate branches
Implementation – Our Setup
Implementation – Our Setup

- Code & bug tracker hosted by GitHub
  - https://github.com/rootjs
Implementation – Our Setup

- Code & bug tracker hosted by GitHub
  - https://github.com/rootjs

- Continuous integration via Jenkins http://jnugh.de:8080/
  - Integration tests
  - Code coverage
  - Doxygen documentation on http://rootjsdocs.jnugh.de/annotated.html
Implementation – Our Setup

Why GitHub?
Implementation – Our Setup

- Why GitHub?
  - Open source
  - Everyone knows how to use it
  - Always available
Implementation – Our Setup

- Why Jenkins?
Implementation – Our Setup

Why Jenkins?

- Originally wanted TravisCI
  - Building ROOT times out
Implementation – Our Setup

- Why Jenkins?
  - Originally wanted TravisCI
    - Building ROOT times out
  - On our own system timeouts don’t matter
    - Jenkins also gets the job done
Implementation – Our Workflow
Implementation – Our Workflow

- New features are developed in separate branches
  - Pull requests are only merged if all tests pass
  - Pull requests tagged “help wanted“ are discussed during weekly meeting
Implementation – Our Workflow

- New features are developed in separate branches
  - Pull requests are only merged if all tests pass
  - Pull requests tagged “help wanted“ are discussed during weekly meeting
- Each bug in the issue tracker is assigned a new branch containing a test for that bug
  - Bug is fixed in that branch
  - When all tests pass it can be merged
Implementation – Testing
Implementation – Testing

- ~4000 lines of code with 77% line coverage
  - Missing lines are error handling or seldom used argument types (eg. ushort)
Implementation – Testing

- ~4000 lines of code with 77% line coverage
  - Missing lines are error handling or seldom used argument types (e.g. ushort)
- 89 tests used in continuous integration at the end of implementation
Implementation – Talking to Node: NodeHandler

rootS:NodeHandler

- exposeROOT()

expose global functions
- exposeGlobalFunctions()

expose global variables
- exposeGlobals()

expose macros
- exposeMacros()

expose classes & namespaces
- exposeNamespaces()
Implementation – Talking to Node: NodeHandler

- V8 provides an exports
  - Expose everything using `Set` on that object
Implementation – Talking to Node: NodeHandler

- V8 provides an exports
  - Expose everything using `Set` on that object
- Use ROOT’s `GetListofGLobals`, `gClassTable` etc.
  - Iterate those lists and create Templates/Proxies
  - Set them as properties in the exports object
Implementation – Talking to Node: NodeHandler

- V8 provides an exports
  - Expose everything using `Set` on that object
- Use ROOT‘s GetListofGlobals, gClassTable etc.
  - Iterate those lists and create Templates/Proxies
  - Set them as properties in the exports object
- How do we make sure ROOT‘s namespaces are preserved?
  - Each namespace gets a template which is `Set` to the export object
  - Classes are `Set` in their respective namespace‘s object

![Diagram of export process]

- `exposeNamespaces`
- `exposeClasses & namespaces`
- `expose Macros`
- `exposeGlobalFunctions`
- `exposeGlobalVariables`
- `exposeROOT`
Implementation – Talking to Node: Callbacks
Implementation – Talking to Node: Callbacks

Each exposed function is associated with a static method in the CallbackHandler
Implementation – Talking to Node: Callbacks

- Each exposed function is associated with a static method in the CallbackHandler
  - Functions “know” whether they are static, a constructor...
  - Can handle them accordingly
Implementation – Factories
Implementation – Factories

- Factories create wrapper proxies for ROOT objects, primitive data and functions
  - Invoked whenever a constructor is called
  - Invoked whenever a function is called for the first time
Implementation – Factories

- Factories create wrapper proxies for ROOT objects, primitive data and functions
  - Invoked whenever a constructor is called
  - Invoked whenever a function is called for the first time
- Template factory creates function templates for classes and namespaces
  - Iterates the class/namespace's ListOfPublicDataMembers etc.
  - Creates proxies for those and sets them as properties in the v8 template it is creating
Implementation – Proxies
Implementation – Proxies
Implementation – Proxies

Correct proxy to be used is selected using cling
Implementation – Proxies

- Correct proxy to be used is selected using cling
- Read/Writes happen in ROOT memory space
  - Everything is in sync all the time
Implementation – Proxies

- Correct proxy to be used is selected using cling
- Read/Writes happen in ROOT memory space
  - Everything is in sync all the time
- Memory addresses come from our MetaInfo implementation
Implementation – Proxies

- Correct proxy to be used is selected using cling
- Read/Writes happen in ROOT memory space
  - Everything is in sync all the time
- Memory addresses come from our MetaInfo implementation

- What about pointers?
  - Or pointer pointers?
  - Or pointer pointer pointers?
Implementation – Proxies

- Correct proxy to be used is selected using cling
- Read/Writes happen in ROOT memory space
  - Everything is in sync all the time
- Memory addresses come from our MetaInfo implementation

- What about pointers?
  - Or pointer pointers?
  - Or pointer pointer pointers?

→ Normalize memory address by referencing/dereferencing until it is a `void**`
Implementation – FunctionProxy
Implementation – FunctionProxy

- Use cling to get function pointers based on call signatures
  - gInterpreter->CallFunc_SetFuncProto
Implementation – FunctionProxy

- Use cling to get function pointers based on call signatures
  - gInterpreter->CallFunc_SetFuncProto

- Parameters are passed using a buffer
  - Scalar values are copied into the buffer (converted from v8 objects)
  - Objects are always passed by address
Implementation – FunctionProxy

- Use cling to get function pointers based on call signatures
  - `gInterpreter->CallFunc_SetFuncProto`

- Parameters are passed using a buffer
  - Scalar values are copied into the buffer (converted from v8 objects)
  - Objects are always passed by address

- Creation of buffer and call of function are separated to support async calling
Implementation – FunctionProxy
Implementation – FunctionProxy

- What was hard:
  - Very little documentation for cling API
  - Had to guess how to use some of the functionality
  - PyROOT was a helpful reference
Implementation – FunctionProxy

- **What was hard:**
  - Very little documentation for cling API
  - Had to guess how to use some of the functionality
  - PyROOT was a helpful reference

- **What we didn't think of:**
  - Overloaded methods that support different types of floating point numbers
    - If number fits into type, overloaded version is selected
    - Problem because for example
      - First variant uses float
      - We have a small number
      - Number has many decimal places
Implementation – Asynchronous Calls
Implementation – Asynchronous Calls

- During design we were uncertain how async calling would work
  - Planned to use ROOT's TThread
Implementation – Asynchronous Calls

- During design we were uncertain how async calling would work
  - Planned to use ROOT’s TThread
- V8 does not work in a multithreaded environment
  - Interactions with node need to be done from main thread
Implementation – Asynchronous Calls → libuv
Implementation – Asynchronous Calls → libuv

- Libuv’s message passing between async workers and v8
Implementation – Asynchronous Calls → libuv

- Libuv’s message passing between async workers and v8
- We use libuv because it integrates great with node
  - No need to wait for threads actively
  - Handled by signals → non-blocking & no waste of CPU time
Implementation – ObjectProxyBuilder
Implementation – ObjectProxyBuilder

- V8 does not work with libuv workers
Implementation – ObjectProxyBuilder

- V8 does not work with libuv workers
- ObjectProxy makes heavy use of v8
Implementation – ObjectProxyBuilder

- V8 does not work with libuv workers
- ObjectProxy makes heavy use of v8
- When running a constructor ObjectProxy uses a v8 FunctionTemplate
  - Can not create ObjectProxies in worker threads
Implementation – ObjectProxyBuilder

- V8 does not work with libuv workers
- ObjectProxy makes heavy use of v8
- When running a constructor ObjectProxy uses a v8 FunctionTemplate
  - Can not create ObjectProxies in worker threads

→ ObjectProxyBuilder contains meta data to be used in the main thread
Implementation – Differences between Proxies

- PointerInfo
- MemberInfo
- GlobalInfo
- EnumInfo
- FunctionInfo

MetaInfo
Implementation – Differences between Proxies

- Interfaces of ROOT classes we have to wrap in a proxy are inconsistent
  - Want to have unified interface for all Proxies
Implementation – Differences between Proxies

- Interfaces of ROOT classes we have to wrap in a proxy are inconsistent
  - Want to have unified interface for all Proxies

- Another layer of indirection saves the day:
  - MetaInfo encapsulates differences
  - Each Proxy instance has a MetaInfo object associated that contains the needed implementations
Implementation – Want more Libraries?
Implementation – Want more Libraries?

- gSystem can load additional shared libraries
  - We have to update our bindings whenever new classes are added
Implementation – Want more Libraries?

- gSystem can load additional shared libraries
  - We have to update our bindings whenever new classes are added
- Provide an additional function `loadlibrary()` and `refreshExports()`
  - Loads a library and updates or just updates respectively
  - Simply reexecutes exposure process
    - Traverses gClassTable etc and adds any new classes, globals ..
    - Fast because v8 properties are stored in a hashtable
  - Allows for library loading during runtime and even creation of new global variables
LIVE DEMONSTRATION
Project Review

- Features
Project Review

- Features
  - Fulfills all required criteria
Project Review

Features

- Fulfills all required criteria
- Runs on Linux and Mac OS X
Project Review

- Features
  - Fulfills all required criteria
  - Runs on Linux and Mac OS X
  - Supports asynchronous execution for all functions
Project Review

Features

- Fulfills all required criteria
- Runs on Linux and Mac OS X
- Supports asynchronous execution for all functions
- Supports C++ operators
Project Review

- Features
  - Fulfills all required criteria
  - Runs on Linux and Mac OS X
  - Supports asynchronous execution for all functions
  - Supports C++ operators
  - Supports loading ROOT libraries
Project Review

Features

- Fulfills all required criteria
- Runs on Linux and Mac OS X
- Supports asynchronous execution for all functions
- Supports C++ operators
- Supports loading ROOT libraries

Open issues
Project Review

Features

- Fulfills all required criteria
- Runs on Linux and Mac OS X
- Supports asynchronous execution for all functions
- Supports C++ operators
- Supports loading ROOT libraries

Open issues

- Use function pointer as return value
Project Review

Features
- Fulfils all required criteria
- Runs on Linux and Mac OS X
- Supports asynchronous execution for all functions
- Supports C++ operators
- Supports loading ROOT libraries

Open issues
- Use function pointer as return value
- Encapsulation of anonymous types
Project Review
Project Review

- Team performance
  - It went really well
    - Especially considering it was the first collaborated software project for most of us
    - Especially considering most of us didn't know any or very little C++ or JavaScript
Project Review

- Team performance
  - It went really well
    - Especially considering it was the first collaborated software project for most of us
    - Especially considering most of us didn't know any or very little C++ or JavaScript

- What could be improved?
Project Review

- Team performance
  - It went really well
    - Especially considering it was the first collaborated software project for most of us
    - Especially considering most of us didn't know any or very little C++ or JavaScript

- What could be improved?
  - Time management
    - Often difficult because of university/work commitments
Project Review

- Team performance
  - It went really well
    - Especially considering it was the first collaborated software project for most of us
    - Especially considering most of us didn't know any or very little C++ or JavaScript

- What could be improved?
  - Time management
    - Often difficult because of university/work commitments
  - Task management
    - Difficult at first to coordinate who does what
    - Got better towards the end with Github issues
Project Review
Project Review

- What we learned
Project Review

- What we learned
  - Git is awesome!
Project Review

- What we learned
  - Git is awesome!
  - LaTeX has a steep learning curve
Project Review

- What we learned
  - Git is awesome!
  - LaTeX has a steep learning curve
  - Testing is effective!
Project Review

What we learned

- Git is awesome!
- LaTeX has a steep learning curve
- Testing is effective!
- A lot about the Google v8 engine
Project Review

- What we learned
  - Git is awesome!
  - LaTeX has a steep learning curve
  - Testing is effective!
  - A lot about the Google v8 engine
  - Old projects may have a somewhat chaotic code base
Questions?

- Find rootJS on github: https://github.com/rootjs
Sources
- Danilo Piparo and Olivier Couet. ROOT Tutorial for Summer Students
- CERN. ROOT application domains
  - https://root.cern.ch/application-domains
- Wiki. Node.js logo
  - https://upload.wikimedia.org/wikipedia/commons/d/d9/Node.js_logo.svg
- exortech. v8 logo
- CERN. ROOT Shower Event Display
  - https://root.cern.ch/rootshower00png
- http://jestingstock.com/image-computer-icon.html
- Axel Naumann. ROOT logo
Sources

- Octodex Github. logo
  - https://octodex.github.com/images/octobiwan.jpg

- Jenkins-CI. jenkins logo
  - https://wiki.jenkins-ci.org/display/JENKINS/Logo

- geekherocomic. The Price Of Continuous Integration

- libuv. libuv logo
  - http://docs.libuv.org/en/v1.x/_static/logo.png