Developer Guide

version **0.7**

Repo: <https://github.com/pointonsoftware/pscore>

CORE Business Software Engine Jira: <https://pointon.atlassian.net/browse/PCOR>

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Description** | **Author** | **Version** |
| 03/15/2020 | First created | benziv | v0.1 |
| 06/28/2020 | Added coding guide | benziv | v0.2 |
| 08/15/2020 | Updated coding guide | benziv | v0.3 |
| 09/17/2020 | Added development setup and tips n tricks | benziv | v0.4 |
| 10/05/2020 | Added How to add a new screen section | benziv | v0.5 |
| 10/10/2020 | Added How to add a new Core module | benziv | v0.6 |
| 10/14/2020 | Added How to create a release | benziv | v0.7 |

**Table of Contents**

1. Development setup
2. Coding Guide
3. Tips and Tricks
4. How to add a screen
5. How to add a new Core module
6. How to create a release
7. Developer workflow

# Development setup

##### Accounts

1. Create an account in <https://www.github.com/join>
2. Create an account in <https://www.atlassian.com/>
   1. Look for the login button.
   2. You can use social media accounts as your login info.
3. Create an account in <https://login.qt.io/register>

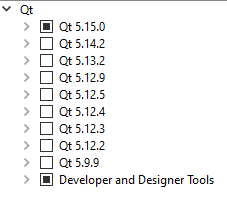
##### Downloads

1. Download Qt from <https://www.qt.io/download-open-source>
2. Download GitHub desktop from https://desktop.github.com

##### Installation

* **Qt** - Note: QT installation may take ~5.70GB of space and needs internet connection.

1. Install Qt and login when needed.
   1. Default installation is in C:/ directory.
2. Make sure to check the following Qt setup:



* 1. **Expand Qt 5.15.0** and select the following:
* MinGW 8.1.0 32-bit
* MinGW 8.1.0 64-bit
* Sources
* Qt Charts
* Qt Data Visualization
  1. **Expand Developer and Designer Tools** and select the following:
* Qt Creator 4.13.0 CDB Debugger Support
* Debugging Tools for Windows
* MinGW 8.1.0 32-bit
* MinGW 8.1.0 64-bit
* CMake 3.17.1 64-bit
* Ninja 1.10.0

##### Cloning the repository

* **GitHub Desktop** -Note: Setup will need internet connection)

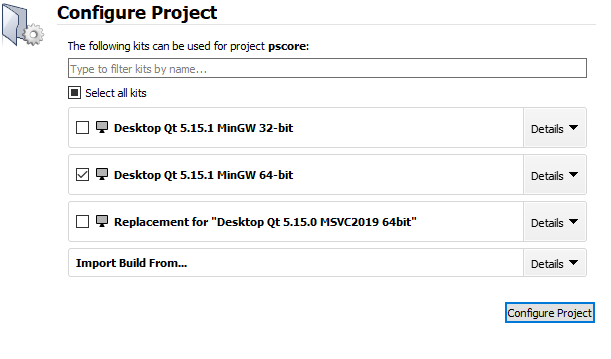
1. Execute the downloaded installer
2. Click on **Sign in to Github.com**
   1. Sign in with your GitHub credentials

* *Before proceeding, make sure you have been added to the GitHub project (contact BenZiv).*

1. In the **Let’s get started** page, select **pointonsoftware/pscore**.
2. Click on **Clone pointonsoftware/pscore**.
3. Set the local path to any directory that you want. Hit **Clone**.

##### Importing project to Qt

1. Run **Qt Creator**.
2. Select File-> Open File or Project
3. Go to the pscore root folder and select **CMakeLists.txt**
4. In Configure Project, select **Desktop MinGW 64-bit** (or depends on your system)



1. Right click on the core project and select **Run CMake**

# Coding Guide

**Namespace:**

Give a namespace to all the code in your module. For example, for a module MyModule.dll, you may give its code the namespace MyModule.

**Functions:**

* Functions should assure callers that even if an exception is thrown, program invariants remain intact (i.e., no data structures are corrupted) and no resources are leaked. Functions should also assure callers that if an exception arises, the state of the program remains as it was prior to the call.
* Use meaningful and related function names.
* If a function is very small and time-critical, declare it inline
* Keep functions short and simple.
  + Should perform a single operation.
  + This way it simpler to understand, test and reuse.

**Example**

Consider:

void read\_and\_print() // bad

{

int x;

cin >> x;

// check for errors

cout << x << "\n";

}

This is a monolith that is tied to a specific input and will never find another (different) use. Instead, break functions up into suitable logical parts and parameterize:

int read(istream& is) // better

{

int x;

is >> x;

// check for errors

return x;

}

void print(ostream& os, int x)

{

os << x << "\n";

}

These can now be combined where needed:

void read\_and\_print()

{

auto x = read(cin);

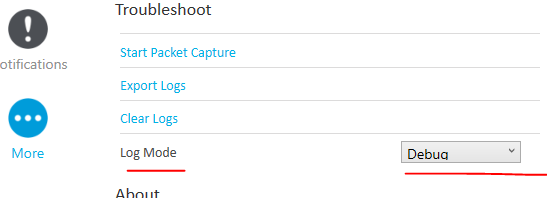
print(cout, x);

}

* For general use, take T\* or T& arguments rather than smart pointers
  + If you won’t assign the pointer argument to any member (i.e. it will only be used by the function), avoid using smart pointer.

**Logging:**

* Write time log for all database request operations.
  + From start of the function call to return of data.
* The logger can be switched to different logger types (file, console, socket).
* LOG macros usage
  + LOG\_DEBUG – for verbose logging
  + LOG\_INFO – use for any useful information that is helpful in knowing what the user was trying to do.
  + LOG\_WARN – use for any conditions where we expect to fail but is handled by the system
  + LOG\_ERROR – use in conditions that are critical and should not fail. Otherwise
* Provide an option in the application for the log mode:
  + Debug, Warn, Info, Error



**Pointers:**

* Use -Werror=return-local-addr compiler flag to detect dangling pointers.

main.cpp:7:7: error: address of local variable 'object' returned [-Werror=return-local-addr]

int object = 10;

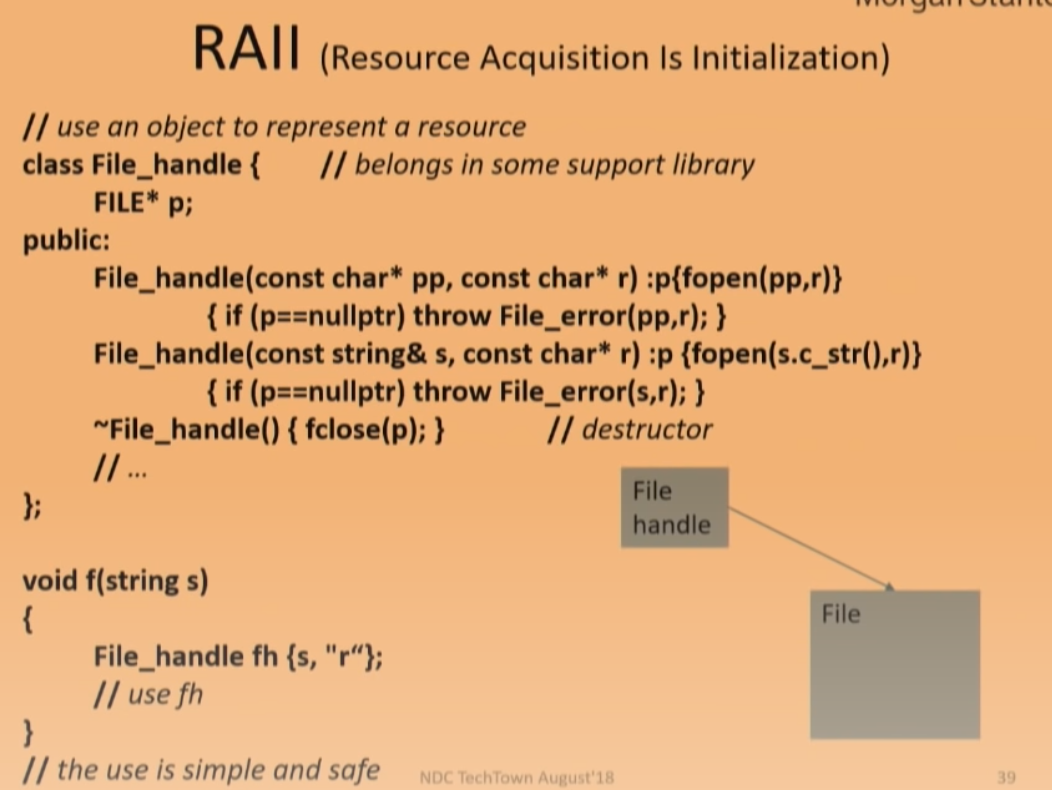
**Const-ness:**

* int \* mutable\_pointer\_to\_mutable\_int;
* int const \* mutable\_pointer\_to\_constant\_int;
* int \*const constant\_pointer\_to\_mutable\_int;
* int const \*const constant\_pointer\_to\_constant\_int;

**Design Patterns:** <https://www.youtube.com/watch?v=j9arNRRoPe8>

* Adapter pattern to extend std::string to accept const char\* null and have to\_lower function.
* Builder pattern for the JSON string builder
* Builder Facet for the Items (ItemBuilder, ItemPriceBuilder, ItemSupplierBuilder).
* Builder Facet for the Person (PersonBuilder, CustomerBuilder, UserBuilder, EmployeeBuilder <- for non-system user);

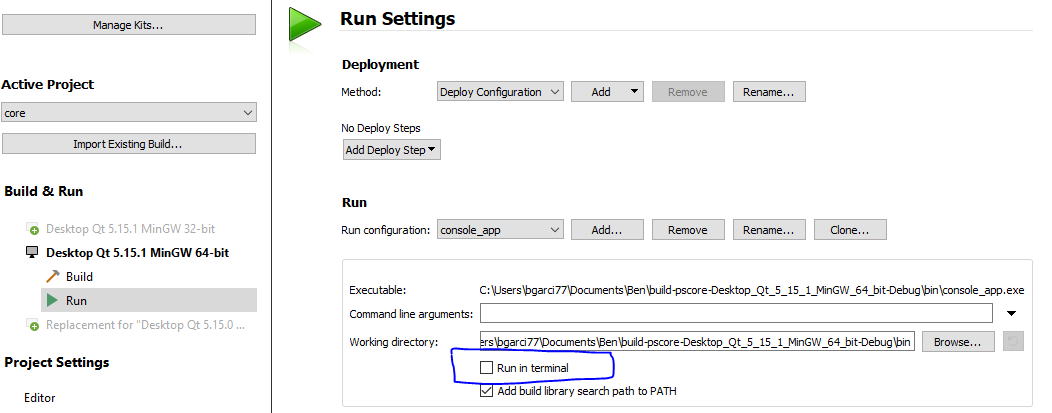
**File Handler:**



# Tips and Tricks

**Console app not showing up in Qt [All platforms]**

* If console application does not show up, make sure to check/enable **Run in Terminal** in **Build & Run** configuration.



**Console app not working in Qt Ubuntu Linux**

* If console application does not show up, this is possibly because **Qt** is not able to work with the current **Linux terminal**.
  + To resolve, go to Tools->Options->Environment->System and select your current terminal.

# Adding a new screen

1. **Create** the .hpp and .cpp file for the new screen in orchestra/application/screen.
   * Make sure to add the copyright notice
   * Make sure to add the correct header guards
     + Format: FOLDERNAME\_FILENAME\_FILEEXTENSION\_
2. **Add** the new files to orchestra/application/CMakeLists.txt
3. **Add** the #include <screeniface.hpp> to your header file
4. **Inherit** from screen::ScreenInterface and **implement** void show(std::promise<screen::display>\* promise) in your new class.
5. **Add** a new enum in [orchestra/application/screen/screendefines.hpp](https://github.com/pointonsoftware/pscore/pull/37/files#diff-69f6eaf4fb2f6f913a916e31a80c7418)  for the new screen
6. In [orchestra/application/flowcontroller.cpp](https://github.com/pointonsoftware/pscore/pull/37/files?file-filters%5B%5D=.cpp&file-filters%5B%5D=.txt#diff-55cca6bfdcc86ccd3961eb2eba4eff2f), **add** the switch-case in FlowController::show() to show your new screen.

**Excellent!** Now your new screen can be transitioned from other screens by specifying your new enum in their promise parameter (i.e. promise->set\_value(screen::display::NEW\_ENUM)).

**Note:** For testing purposes, you can replace screen::display nextScreen = screen::display::LOGIN; with your screen enum to set it as the first screen. However, keep in mind that some screens require special initialization (e.g. using screen shared data).

# Adding a new Core module

**API creation**

1. **Create** the module folder inside core/domain.
2. **Create** a CMakeLists.txt file and **specify** the following:
   * Project name
   * Library name and type
3. **Create** the interface folder inside core/domain/[module\_folder].
4. **Create** the controller, data and view interfaces inside the interface folder.
5. **Add** the new files to the CMakeLists.txt of the new module.
6. **Add** the new module as subdirectory in core/ CMakeLists.txt file.

**Controller**

1. **Create** the .hpp and .cpp controller files for the new module in core/domain/[module\_folder].
2. **Inherit** from controller interface and **implement** the pure virtual functions in your new controller.

**View**

1. **Add** the module screen.
   * **See** *Adding a new screen* section of this document.
2. **Inherit** from module view interface and **implement** the pure virtual functions in your new screen.

**Data Provider**

1. **Create** the .hpp and .cpp file for the new data provider in orchestra/datamanager.
2. **Add** the new files to orchestra/datamanager/CMakeLists.txt
3. **Add** the #include <domain/[module\_folder]/interface/[module]dataif.hpp> to your header file
4. **Inherit** from [module]dataInterface and implement the functions.

**Voila!** You’re all set to finish the implementation.

**Note:** For all created files, make sure to:

* + **Add** the copyright notice
  + **Add** the correct header guards
    - Format: FOLDERNAME\_FILENAME\_FILEEXTENSION\_

# Creating a release

1. **Create** the latest master.
2. **Do** git log --pretty=%s vx.x.x..HEAD - *where vx.x.x is the current version.*
   * Copy the result, we will need it for the next steps.
3. **Create** a tag
   * git tag vMAJOR.MINOR.REV(.HOTFIX) - *e.g.* git tag v1.0.0
4. **Push** the tag
   * git push origin vx.x.x - *where vx.x.x is the new version.*
5. **Go to** *github.com -> code -> tags* and **find** the *tag* that you just created.
6. On the tag page, **click** *Edit release*.
7. **Paste** the result from step 2 and **arrange** according to the following format:
   * ### Upgrade Steps

- [ACTION REQUIRED]

-

### Breaking Changes

-

### New Features

-

### Bug Fixes

-

### Improvements

-

### Other Changes

-

1. **Click** on Update release
   * Note: **Tick** the *This is a pre-release* checkbox when applicable.