
CARTA Viewer Documentation

Release 0.4

CARTA Team

Mar 20, 2017

Contents

1	User Guide	1
1.1	CARTA User Guide	1
2	Build Guide	3
2.1	How to compiler CARTA	3
3	Developer Guide	9
3.1	Writing an image plugin for CARTA in C++	9
3.2	Setting up a build environment	9
3.3	Writing the plugin	9
3.4	Appendix A : setting up the build environment	10
3.5	Appendix B : how plugins work in CARTA	12
3.6	Appendix C : plugin structure	15
3.7	Appendix D : plugin.json	15
3.8	Appendix E: CARTA config file	16
3.9	Appendix F: libraries & versions	17
4	C++ Auto Generated Documentation	19
4.1	CPP Auto Generated Documentation	19
5	Python Auto Generated Documentation	21
5.1	Python Auto Generated Documentation	21
6	Indices and tables	23

CARTA User Guide

Guide will be inserted here, still works in progress...

How to compiler CARTA

Introduciton

At the present time we have successfully compiled CARTA viewer on the following platforms - **Ubuntu 14.04**, **Cent OS 6.7** and **Mac OS X 10.10**. It is possible to compile and build CARTA viewer on other UNUIX like OSes, however these are the officially supported operating systems for CARTA deployment. This guide will provide detail instructions on how to build CARTA viewer for the above platforms.

Downloading CARTA source code

CARTA source code is publicly available on the git hub via following [CARTA](#) github link. It can be cloned using:

```
$git clone https://github.com/Astroua/CARTAviS
```

You have to make sure that you have version git client higher than 1.7 (this can be an issue with **CentOS 6.7**). It should be cloned into some directory on your build machine, which will be referred further in this document as `$CARTAROOT`. In the example below `$CARTAROOT` is set to `$HOME/src/CARTA`.



```
alex-s-imac:~/src/CARTA$ls
CARTA_0.4.1.app/          Cython-0.22/
CARTAviS/                 Dockerfile
CARTAviS-externals/      Images/
CartaTemplate/           PureWeb/
alex-s-imac:~/src/CARTA$
```

Software required to build on Ubuntu 14.04

Installing development tools and required libraries

To compile CARTA C++ compiler that supports C++ 11 standards is required. On Ubuntu 14.04 we are using g++ 4.8.2. It can be installed using following command:

```
$sudo apt-get install -y build-essential
```

Other development tools such as *git*, *autoconf*, *automaker*, *cmake* need to be installed as well via *apt-get*. Ubuntu 14.04 comes with *Python 2.7* installed, if it is not on the system it has to be installed as well along with *NumPy*, *Astropy* and *Cython* python modules. Following development libraries are required to be present *mesa-common-dev* and *libglul-mesa-dev*.

Installing QT library

Open source Qt Framework & Qt Development Tools can be downloaded from <http://www.io.qt> web site. At the present time we are using qt version 5.3 for CARTA development. Version 5.3 is not the latest one but it is available for download from qt web site and should be used. There is QT graphical installer for linux/Mac OSX platforms.

Installing CASACORE library

CASACORE library is available on [CASACORE github](#) and can be cloned using following command:

```
$git clone https://github.com/casacore/casacore
```

There are number of required packages that are needed to be installed before CASACORE library it can be compiled. To be able to use CASACORE library CASACORE *data* package needs to be available on the system (also known as casacore measures data). It is usually installed in @HOME/data directory or in /usr/local/geodetic and /usr/local/ephemerides directories.

Installing third party libraries

We need to create directory with soft links to all third party libraries in order to successfully build CARTA. Here are instructions on how to do that:

```
$cd $CARTAROOT
$mkdir -d CARTAvis-externals/ThirdParty
$cd CARTAvis
$ln -s ../CARTAvis-externals Externals
$ln -s ./Externals/ThirdParty ThirdParty
```

Inside ThirdParty directory there will be links to locations where all third party libraries are located. Following libraries/packages are required to be installed and soft links to their locations created:

qwt-6.1.3 - graphics extension to the Qt GUI application framework

gooxdoo-3.5 - this is not the latest version but CARTA will not work with latest version, it is very important to install 3.5 version.

cfitsio-3.360

wcslib-4.23

pureweb-4.1.1

ast-8.0.2

casacore - link to casacore library that has been compiled

In the directory `$CARTAROOT/CARTAVIS/carta/scripts` there is script `createlinks.sh` that will create soft links inside `ThirdParty` directory. It needs to be edited and adjusted to reflect locations of installed packages on the target system.

Software required to build on Cent OS 6.7

Installing development tools and required libraries

Installing QT library

Open source Qt Framework & Qt Development Tools can be downloaded from <http://www.io.qt> web site. At the present time we are using qt version 5.3 for CARTA development. Version 5.3 is not the latest one but it is available for download from qt web site and should be used. There is QT graphical installer for linux/Mac OSX platforms.

Installing CASACORE library

CASACORE library is available on [CASACORE](#) github and can be cloned using following command:

```
$git clone https://github.com/casacore/casacore
```

There are number of required packages that are needed to be installed before CASACORE library it can be compiled. To be able to use CASACORE library CASACORE *data* package needs to be available on the system (also known as casacore measures data). It is usually installed in `@HOME/data` directory or in `/usr/local/geodetic` and `/usr/local/ephemerides` directories.

Installing third party libraries

We need to create directory with soft links to all third party libraries in order to successfully build CARTA. Here is instructions on how to do that:

```
$cd $CARTAROOT
$mkdir -d CARTAVIS-externals/ThirdParty
$cd CARTAVIS
$ln -s ../CARTAVIS-externals Externals
$ln -s ../Externals/ThirdParty ThirdParty
```

Inside `ThirdParty` directory there will be links to locations where all third party libraries are located. Following libraries/packages are required to be installed and soft links to their locations created:

qwt-6.1.3 - graphics extension to the Qt GUI application framework

gooxdoo-3.5 - this is not the latest version but CARTA will not work with latest version, it is very important to install 3.5 version.

cfitsio-3.360

wcslib-4.23

pureweb-4.1.1

ast-8.0.2

casacore - link to casacore library that has been compiled

In the directory `$CARTAROOT/CARTAviS/carta/scripts` there is script `createlinks.sh` that will create soft links inside `ThirdParty` directory. It needs to be edited and adjusted to reflect locations of installed packages on the target system.

Software required to build on Mac OSX 10.10

Installing development tools and required libraries

Installing QT library

Open source Qt Framework & Qt Development Tools can be downloaded from <http://www.io.qt> web site. At the present time we are using qt version 5.3 for CARTA development. Version 5.3 is not the latest one but it is available for download from qt web site and should be used. There is QT graphical installer for linux Ubuntu platform.

Installing CASACORE library

CASACORE library is available on [CASACORE github](#) and can be cloned using following command:

```
$git clone https://github.com/casacore/casacore
```

There are number of required packages that are needed to be installed before CASACORE library it can be compiled. To be able to use CASACORE library CASACORE data package needs to be available on the system (also knows as casacore measures data). It is usually installed in `@HOME/data` directory or in `/usr/local/geodetic` and `/usr/local/ephemerides` directories.

Installing third party libraries

We need to create directory with soft links to all third party libraries in order to successfully build CARTA. Here is instructions on how to do that:

```
$cd $CARTAROOT
$mkdir -d CARTAviS-externals/ThirdParty
$cd CARTAviS
$ln -s ../CARTAviS-externals Externals
$ln -s ../Externals/ThirdParty ThirdParty
```

Inside `ThirdParty` directory there will be links to locations where all third party libraries are located. Following libraries/packages are required to be installed and soft links to their locations created:

qwt-6.1.3 - graphics extension to the Qt GUI application framework

qoofdoo-3.5 - this is not the latest version but CARTA will not work with latest version, it is very important to install 5.3 version.

cfitsio-3360

wcslib-4.23

pureweb-4.1.1

ast-8.0.2

casacore - link to casacore library that has been compiled

In the directory `$CARTAROOT/CARTAviS/carta/scripts` there is script `createlinks.sh` that will create soft links inside `ThirdParty` directory. It needs to be edited and adjusted to reflect locations of installed packages on the target system.

Generating CARTAvis html files

Before each compile javascript files need to be re-generated. Follow example below to re-generate javascript files:

```
$cd $CARTAROOT/CARTAvis/carta/html5/common/skel
$./generate.py source
$./generate.py
```

This is very important step, skipping it might not break compile process but will results in very unusual run time behaviour of CARTAvis.

Compiling CARTAvis desktop binaries

Instead of compiling executable in the same directory where source code is, we use separate build directory. Change your working directory to the location where you want to create build directory and create build directory as in the example below

```
alex-s-imac:~/src/CARTA$ls
CARTA_0.4.1.app/          Cython-0.22/
CARTAvis/                Dockerfile
CARTAvis-externals/     Images/
CartaTemplate/           PureWeb/
alex-s-imac:~/src/CARTA$mkdir build
alex-s-imac:~/src/CARTA$cd build
alex-s-imac:~/src/CARTA/build$
```

Now issue *qmake* command to create all make files:

```
$ qmake NOSERVER=1 CARTA_BUILD_TYPE=dev ~/src/CARTA/CARTAvis/carta -r
```

In this example `~/src/CARTA/CARTAvis/carta` is the location where *CARTA* source code is located, it has to be adjusted accordingly if it is different from the example above. After all make files are generated source code can be compiled with make command as in the following example:

```
$make -j4
```

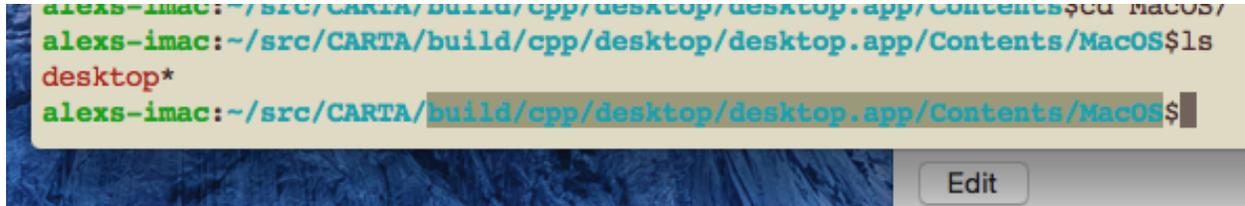
Optionally *-jN* parameter can be used to speed up compile process with *N* simultaneous compiles. *N* parameter should not exceed number of processors available on the build machine as this will not have any effect.

Compiling CARTAvis server binaries

Compiling server binaries is the same as desktop one except when issuing make command *NOSERVER* option should be set to 0 or omitted: `$ qmake NOSERVER=1 CARTA_BUILD_TYPE=dev ~/src/CARTA/CARTAvis/carta -r` Server executable will be compiled in `cpp/server` directory, relative to the build directory.

Running CARTAvis viewer after successful build

After *CARTA* viewer has been successfully compiled its binary can be found in `build/cpp/desktop` directory on CentOS and Ubuntu platforms and in `build/cpp/desktop/desktop.app/Contents/MacOS` on Mac OS X, screenshot below shows location of CARTAvis binary on Mac OS X.



Name of the executable is “desktop”, it can be executed using following command:

```
./desktop --html ~/src/CARTA/CARTAvis/carta/VFS/DesktopDevel/desktop/desktopIndex.  
→html --scriptPort 9999
```

Third parameter is the location of `desktopIndex.html` file, usually points to the place where source code has been downloaded from git repository. In the example above *CARTA* source code has been placed in `~/src/CARTA` (pointing to by variable `$CARTAROOT`) directory. Optionally additional parameter can be passed on the command line to specify image file to be open by the viewer on a startup, as in the example below:

```
./desktop --html ~/src/CARTA//CARTAvis/carta/VFS/DesktopDevel/desktop/desktopIndex.  
→html --scriptPort 9999 ~/CARTA/Images/555wmos.fits
```

Writing an image plugin for CARTA in C++

The purpose of this document is to describe how to write a C++ plugin that allows opening images of new file formats in *CARTA*.

1. Setting up build environment
2. Writing the plugin
3. Testing the plugin

While reading this guide, please refer to appendices B, C and D, which contain some general information about the plugin architecture in *CARTA*.

Setting up a build environment

At the moment we do not have an easy mechanism to set up a build environment for just compiling and testing plugins. That means for now you will need to set up your computer to build all of *CARTA*. In the future we hope to have a simplified setup just to be able to build plugins. Please follow the instructions in Appendix A on how to setup your machine to be able to compile *CARTA* and your plugin.

Writing the plugin

The easiest way to write a new plugin for *CARTA* is to take an existing one, clone it and modify it to do what you need. For adding a new image format the best plugin to clone is the *qimage* plugin, e.g.:

```
> cd /tmp/xxx/CARTAVIS/carta/cpp/plugins
> cp -a qimage myimage
> cd myimage
> mv qimage.pro myimage.pro
```

```
... rename the header/sources as you see fit
> emacs plugin.json
... fix up entries to match your plugin
> emacs ../plugins.pro
... add myimage to the list of plugins
> cd ..
> make
```

At this point you should have a functional plugin which does exactly the same thing qimage plugin does. It is now time to modify the sources to make it do what you want. The best way to go about that is to understand what the existing code does.

The following interface classes are of particular interest, as these are the ones you may need to implement for your plugin to work. They should have sufficient in-code documentation to get you started:

```
IPlugin
Carta::Lib::NdArray::RawViewInterface
CoordinateFormatterInterface
Carta::Lib::Image::MetaDataInterface
Carta::Lib::Image::ImageInterface
Carta::Core::Hooks::LoadAstroImage
```

The main class (the one that implements the plugin), looks like this (qimage/QImagePlugin.h):

```
/// This plugin can read image formats that Qt supports.
#pragma once
#include "CartaLib/IPlugin.h"
#include <QObject>
#include <QString>
class QImagePlugin : public QObject, public IPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID "org.cartaviewer.IPlugin")
    Q_INTERFACES( IPlugin)
public:
    QImagePlugin(QObject *parent = 0);
    virtual bool handleHook(BaseHook & hookData) override;
    virtual std::vector<HookId> getInitialHookList() override;
};
```

Study the implementation of these methods (qimage/QImagePlugin.cpp).

Appendix A : setting up the build environment

Here we try to explain how to setup your machine to be able to build CARTA from sources. You will most likely need to install some prerequisites first, then download the CARTA sources and compile them.

CARTA prerequisites

Operating system

We highly recommend you do your development on Ubuntu 14.04 LTS 64-bit. The instructions in this document are written for this OS.

Compiler

You will need g++ version 4.8.1 or newer. If you have not installed it already, you can:

```
> sudo apt-get install g++
```

Qt

Make sure you have qt installed, at least 5.3. For example:

```
> sudo apt-get install qtbase5-dev
```

Casacore

You will need casacore 2.0.1 or newer to be able to compile all of CARTA. This can be a lengthy procedure, please read the documentation here:

<https://github.com/casacore/casacore>

CARTA's core does not actually depend on casacore, only some of the bundled plugins do. Therefore it's possible to adjust the build process so that you don't need to install casacore at all. This is done by disabling the bundled plugins that require casacore:

- WcsPlotter
- Histogram
- CasaImageLoader
- CasaCore-2.0.01

You can disable them by editing plugins.pro and removing the corresponding entries for these plugins.

Python

If you want python plugins to work, you will need to install python dev files:

```
> sudo apt-get install libpython2.7-dev
```

You can disable python support by disabling the python273 plugin (you have to edit plugins.pro).

QWT

You will need libqwt version 6.1.2 or above. You need to make sure that the qwt you use is compiled against the same version of Qt that you will be using! This is very important. You can get installation instructions here:

<http://qwt.sourceforge.net/>

Qooxdoo

You need to install qooxdoo version 3.5.1 from:

<http://qooxdoo.org/downloads>

Please don't install a newer version, as it will likely not work properly. At the time of writing this document, the following link works:

<http://sourceforge.net/projects/qooxdoo/files/qooxdoo-current/3.5.1/qooxdoo-3.5.1-sdk.zip/download>

Download CARTA sources:

Next you need to download the sources for CARTA from github. We recommend you create an empty directory and do all your work there. For example below we create `/tmp/xxx`:

```
> mkdir /tmp/xxx
> cd /tmp/xxx
> github clone git@github.com:Astroua/CARTAvis.git
  or use Pavol's branch:
> github clone git@github.com:pfederl/CARTAvis.git
```

Next you need to configure CARTA.

- create `./CARTAvis-externals` and setup links
- edit `common_config.pri`

Compile CARTA:

```
> cd /tmp/xxx
> mkdir build
> cd build
> qmake CARTA_BUILD_TYPE=dev /tmp/xxx/CARTAvis/carta/carta.pro
```

Run qooxdoo compiler:

```
> cd /tmp/xxx/CARTAvis/carta/html5/common/skel/
> ./generate.py
```

Compile C++:

```
> cd /tmp/xxx/build
> make
```

Try to run the desktop application:

```
> ./cpp/desktop/desktop --html /tmp/xxx/CARTAvis/carta/VFS/DesktopDevel/desktop/
↳ desktopIndex.html /tmp/some.fits
```

Appendix B : how plugins work in CARTA

The main purpose of a plugin is to extend some aspect of the core viewer functionality. For example, the core could implement some basic colormaps, and a plugin could be used to add additional ones. Or the core could support FITS and CASA input image formats, while a plugin could add support for HDF5. A plugin could also be used to replace/override existing functionality, for example changing the formatting or the type of information present in the status bar. The support for a particular extension will need to be coded into the core. To illustrate how this might work, consider the pseudo code that loads an image from a file. Below is what the implementation might look like without any plugin support:

```

IImage * loadImage( const string & fileName):
    IImage * result = loadFITSorCASA( fileName);
    if( result) return result;
    reportError( "Could not load image...");
    return nullptr; // or throw exception
}

```

The above code handles only FITS and CASA images. If neither can be used, the core gives up and reports an error to the user. To add support for additional image formats, we could augment the above code like this:

```

IImage * loadImage( const string & fileName):
    IImage * result = loadFITSorCASA( fileName);
    if( result) return result;
    // core could not handle this format, let's see if one of the plugins can
    for( auto plugin : allPlugins_that_implement_loadImage) {
        result = plugin-> loadImage( fileName);
        if( result) return result;
    }
    reportError( "Could not load image...");
    return nullptr; // or throw exception
}

```

By adding couple of lines of (pseudo) code, we can now extend supported image formats using plugins. The plugin method 'loadImage' is something I like to refer as a hook (vaguely motivated by <http://en.wikipedia.org/wiki/Hooking>). Hooks are strategically placed pieces of code throughout the core, which call one or more plugins implementing that particular hook. Some hooks are only called once (for example, there is a hook that is executed when the viewer starts, and another when the viewer exits). Other hooks are executed multiple times (e.g. user is trying to load an image with an unknown format). Some hooks have input parameters, others don't. Some hooks have results, some don't. Some hooks are always executed by all plugins that implement them. Some hooks are only executed by the first plugin that returns a valid result.

Since the core of the viewer is written in C++, we decided we'll support plugins written in C++ (for maximum performance). Each plugin is essentially a C++ code compiled into a shared library. We use Qt's lower-level plugin mechanism to gain some platform independence (<http://qt-project.org/doc/qt-5/plugins-howto.html>). Each plugin implements roughly the following interface:

```

class IPlugin {
public:
    virtual std::vector<HookId> getInitialHookList() = 0;
    virtual bool handleHook( BaseHook & hookData) = 0;
};

```

After a plugin is loaded, it's 'getInitialHookList()' method is invoked, which tells the core the list of 'hooks' the plugin implements. This is used as an optimization technique, so that we don't call every single plugin for every hook. The second method 'handleHook()' is called when the core is executing a hook. The name of the hook, it's parameters and the result are all encoded in the single parameter 'hookData' of an abstract type BaseHook. Each hook in the core is essentially a subclass of BaseHook. The BaseHook class is very simple:

```

class BaseHook {
public:
    explicit BaseHook( const HookId & id) : m_hookId(id) {}
    /// dynamic ID
    HookId hookId() const { return m_hookId; }
protected:
    HookId m_hookId;
};

```

The only thing of note here is the HookId type, which is currently an integer. We use these IDs to quickly decide whether we want to do downcast to the proper hook type.

And here is an example of how the loadImage hook might be implemented:

```
class LoadImageHook : public BaseHook {
public:
    typedef IImage * ResultType;
    struct Params {
        Params( QString p_fileName) {
            fileName = p_fileName;
        }
        QString fileName;
    };
    enum { StaticHookId = 5 };
    LoadImage(Params * pptr) : BaseHook( StaticHookId), paramsPtr( pptr) {}
    ResultType result;
    Params * paramsPtr;
};
```

Here is pseudo-code for a C++ plugin that implements only the loadImage hook:

```
class QImagePlugin : public IPlugin {
    virtual std::vector<HookId> getInitialHookList() override;
    virtual bool handleHook(BaseHook & hookData) override;
};
std::vector<HookId> QImagePlugin::getInitialHookList() {
    return { LoadImageHook::StaticHookId };
}
bool MyPlugin::handleHook(BaseHook & hookData) {
    if( hookData.hookId() == LoadImageHook::StaticHookId) {
        LoadImageHook & hook = static_cast<LoadImageHook &>( hookData);
        hook.result = readSomeFileFormatForExampleHDF5( hook.paramsPtr->fileName);
        return hook.result != nullptr;
    }
    warning << "Sorry, don't know how to handle this hook" << hookData.hookId();
    return false;
}
```

The plugins will be loaded and initialized in the same process/thread as the core viewer. All communication (via handleHook) will also be called from the same thread. If they want, plugins can spawn additional threads or processes, but then it's up to them to set up synchronization with the core thread. The core won't provide any such functionality.

At startup all subdirectories of the main 'plugins' directory are scanned (by reading and parsing the plugin.json files). All plugins are compiled into a list. The list is then topologically sorted using the specified dependencies. Then each plugin from the sorted list is processed:

- **we attempt to load the plugin as native (type=c++ | lib)**
 - first we load all libraries under plugin's /lib/ directory (matching *.so and *.so.* patterns)
 - **the loading order of these is determined heuristically, by repeating these steps:**
 - * if the list of libraries to load is empty, break out of this loop
 - * try the libraries on the list that have not been loaded and load them
 - * if a library was loaded, mark it as loaded
 - * if any library was loaded, continue the loop
 - * if no library was loaded, break out indicating a warning

- if the plugin’s type is c++, we load plugin.so, or report error
- if the plugin was not loaded as native, we attempt to load it using loadPlugin hook, i.e. using some other plugin
- if the plugin was loaded (either as native or foreign), we execute it’s initialize method

Appendix C : plugin structure

Each carta plugin lives in its own directory, with the following contents:

plugin.json	required file	meta-information about the plugin
libplugin.so	optional file	the actual C++ plugin
libs/	optional subdirectory	directory of libraries the plugin provides and/or uses
libs/lib1.so ...	optional files	the actual libraries the plugin provides/uses. These can be organized in additional subdirectories. Any .so file will be treated as a provided library and will be loaded (if possible).
other files or subdirectories	optional	anything else the plugin needs?

Appendix D : plugin.json

api:	required integer	which plugin-API does this plugin use
name:	required string	Unique name by which the plugin will be identified. It is a good idea to name the plugin’s directory the same as this name.
version:	required string	version as a string in x.y.z format
type:	required string	type of plugin, currently supported types are: “C++” and “libs”. Other types can be added via plugins (e.g. “python”).
depends:	optional array of strings	array of other plugins that must be present for this plugin to work
description:	optional (ascii string)	short description of the plugin’s functionality
about:	optional (html string)	any other info about the plugin (e.g. home page, authors, organization, etc)
provides: not implemented yet	string list	what does the plugin provide, can be used as dependencies
other:	json object	configuration options for the plugin, not parsed by core. The plugin needs to parse it itself

Example plugin.json file:

```
{
  "api"       : "1",
  "name"      : "qimage",
  "version"   : "1",
  "type"      : "C++",
  "description": "Adds ability to load jpeg/png images. (anything Qt supports)",
  "about"     : "Part of carta. Written by Pavol",
}
```

```
"depends"    : [ ]
}
```

Appendix E: CARTA config file

The configuration file is used to change some settings that are installation wide, for example the search paths for plugins. The location of the main configuration file is by default in:

```
$HOME/.cartavis/config.json
```

This location can be changed in two ways:

- via a command line switch passed to the executable, e.g.

```
--config /tmp/myconfig.json
```

note that for the server-side viz this would require modifying the appropriate plugin.xml file in the PureWeb installation

- via an environment variable called CARTAVIS_CONFIG_FILE, e.g:

```
setenv CARTAVIS_CONFIG=/tmp/myconfig.json
```

If both options are used, the command line takes precedence.

What's in the config file? For example search paths for plugins:

```
{
  "pluginDirs": [
    "$(APPDIR)/../plugins",
    "$(HOME)/.cartavis/plugins"
  ], ...
}
```

The above config.json tells carta to look for plugins in two places:

- relative to the executable
- relative to the user's home directory

You can also disable some plugins:

```
{
  "disabledPlugins" : [
    "ColormapsPy",
    "Colormaps1"
  ],
}
```

Or provide particular plugins some settings:

```
{
  "plugins": {
    "CyberSKA": {
      "vizmanUrl1": "http://localhost:1234/testDir/%%",
      "vizmanUrl2": "http://localhost:8080/test.txt"
    },
    "DevIntegration" : {
      "enabled" : true
    }
  }
}
```

```

    },
  }
}

```

Appendix F: libraries & versions

Here is a list of libraries and their versions that we currently use in development:

qoox-doo	3.5
qwt	6.1.2
qt	5.3 and 5.4
ast lib	8.0.2
casacore	2.0.1
pureweb	4.1.1
wcslib	4.23
cfitsio	3360
rapidjson	from github: https://github.com/drewnoakes/rapidjson/commit/8307f0f4c9035bd63853bdf9e1b951e8c0e37b62

C++ Auto Generated Documentation

CPP Auto Generated Documentation

Python Auto Generated Documentation

The animator module

The cartaview module

The histogram module

The cartavis module

The colormap module

The image module

The layer1 module

The layer2 module

The layer3 module

The snapshot module

The tagconnector module

CHAPTER 6

Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)