

tufao

1.4.1

Generated by Doxygen 1.8.14

Contents

1	Main Page	1
1.1	Introduction	1
1.2	Creating a Tufão-based application	1
1.2.1	Hello World!	2
1.2.2	Revisiting Hello World!	3
1.2.3	And then, there was the code	3
1.2.4	Build system integration	4
2	Design Model	5
3	Tufão's plugin system	7
3.1	Understanding by example	7
3.1.1	The Tufao::HttpPluginServer class	7
3.1.2	The plugin	8
3.1.3	The glue!	8
4	Safe signals	9
4.1	Why some signals are unsafe	10
4.2	Conclusion	10
5	Namespace Index	11
5.1	Namespace List	11
6	Hierarchical Index	13
6.1	Class Hierarchy	13

7	Class Index	15
7.1	Class List	15
8	File Index	17
8.1	File List	17
9	Namespace Documentation	19
9.1	Tufao Namespace Reference	19
9.1.1	Detailed Description	21
9.1.2	Enumeration Type Documentation	21
9.1.2.1	HttpResponseStatus	21
9.1.2.2	WebSocketError	22
9.1.2.3	WebSocketMessageType	23
9.1.3	Function Documentation	23
9.1.3.1	operator<<()	23
10	Class Documentation	25
10.1	Tufao::AbstractHttpServerRequestHandler Class Reference	25
10.1.1	Detailed Description	26
10.1.2	Member Function Documentation	26
10.1.2.1	handleRequest()	26
10.1.2.2	operator std::function< bool()	27
10.2	Tufao::AbstractHttpUpgradeHandler Class Reference	27
10.2.1	Detailed Description	29
10.2.2	Member Function Documentation	29
10.2.2.1	handleUpgrade()	29
10.2.2.2	operator std::function< bool()	30
10.2.2.3	operator std::function< void()	30
10.3	Tufao::AbstractMessageSocket Class Reference	31
10.3.1	Detailed Description	33
10.3.2	Constructor & Destructor Documentation	33
10.3.2.1	AbstractMessageSocket()	33

10.3.3	Member Function Documentation	33
10.3.3.1	connected	33
10.3.3.2	disconnected	34
10.3.3.3	newMessage	34
10.3.3.4	sendMessage	34
10.4	Tufao::ClassHandler Class Reference	35
10.4.1	Detailed Description	36
10.5	Tufao::ClassHandlerManager Class Reference	38
10.5.1	Detailed Description	40
10.5.2	Constructor & Destructor Documentation	40
10.5.2.1	ClassHandlerManager()	40
10.5.3	Member Function Documentation	41
10.5.3.1	addPluginLocation()	41
10.6	Tufao::ClassHandlerPluginInfo Struct Reference	41
10.6.1	Detailed Description	42
10.7	Tufao::Headers Struct Reference	42
10.7.1	Detailed Description	43
10.7.2	Member Function Documentation	43
10.7.2.1	fromDateTime()	44
10.7.2.2	toDateTime()	44
10.8	Tufao::HttpFileServer Class Reference	45
10.8.1	Detailed Description	47
10.8.2	Constructor & Destructor Documentation	48
10.8.2.1	HttpFileServer()	48
10.8.3	Member Function Documentation	48
10.8.3.1	bufferSize()	48
10.8.3.2	canHandleRequest() [1/2]	48
10.8.3.3	canHandleRequest() [2/2]	49
10.8.3.4	handler()	49
10.8.3.5	handleRequest	49

10.8.3.6	serveFile() [1/2]	50
10.8.3.7	serveFile() [2/2]	50
10.8.3.8	setDir()	50
10.9	Tufao::HttpPluginServer Class Reference	51
10.9.1	Detailed Description	53
10.9.2	Constructor & Destructor Documentation	53
10.9.2.1	HttpPluginServer() [1/2]	53
10.9.2.2	HttpPluginServer() [2/2]	53
10.9.3	Member Function Documentation	54
10.9.3.1	config()	54
10.9.3.2	handleRequest	54
10.9.3.3	setConfig()	54
10.10	Tufao::HttpServer Class Reference	57
10.10.1	Detailed Description	60
10.10.2	Member Typedef Documentation	61
10.10.2.1	UpgradeHandler	61
10.10.3	Constructor & Destructor Documentation	61
10.10.3.1	HttpServer()	61
10.10.4	Member Function Documentation	62
10.10.4.1	checkContinue()	62
10.10.4.2	close	62
10.10.4.3	defaultUpgradeHandler()	62
10.10.4.4	handleConnection()	63
10.10.4.5	incomingConnection()	63
10.10.4.6	listen()	63
10.10.4.7	requestReady	63
10.10.4.8	serverPort()	64
10.10.4.9	setTimeout()	64
10.10.4.10	setUpgradeHandler()	65
10.11	Tufao::HttpServerPlugin Class Reference	65

10.11.1 Detailed Description	66
10.11.2 Member Function Documentation	66
10.11.2.1 createHandler()	67
10.12Tufao::HttpServerRequest Class Reference	67
10.12.1 Detailed Description	69
10.12.2 Constructor & Destructor Documentation	70
10.12.2.1 HttpServerRequest()	70
10.12.3 Member Function Documentation	70
10.12.3.1 close	70
10.12.3.2 customData()	70
10.12.3.3 data	71
10.12.3.4 end	71
10.12.3.5 headers() [1/2]	71
10.12.3.6 headers() [2/2]	72
10.12.3.7 method()	72
10.12.3.8 readBody()	73
10.12.3.9 ready	74
10.12.3.10responseOptions()	74
10.12.3.11resume	75
10.12.3.12setCustomData()	75
10.12.3.13setTimeout()	75
10.12.3.14setUrl()	76
10.12.3.15socket()	76
10.12.3.16trailers()	76
10.12.3.17upgrade	77
10.12.3.18url()	77
10.13Tufao::HttpServerRequestRouter Class Reference	78
10.13.1 Detailed Description	80
10.13.2 Member Typedef Documentation	81
10.13.2.1 Handler	81

10.13.3 Constructor & Destructor Documentation	81
10.13.3.1 <code>HttpServerRequestRouter()</code>	81
10.13.4 Member Function Documentation	81
10.13.4.1 <code>handleRequest</code>	82
10.13.4.2 <code>map()</code> [1/2]	83
10.13.4.3 <code>map()</code> [2/2]	83
10.13.4.4 <code>unmap()</code>	83
10.14 <code>Tufao::HttpServerResponse</code> Class Reference	84
10.14.1 Detailed Description	86
10.14.2 Member Enumeration Documentation	87
10.14.2.1 <code>Option</code>	87
10.14.3 Constructor & Destructor Documentation	87
10.14.3.1 <code>HttpServerResponse()</code>	87
10.14.4 Member Function Documentation	88
10.14.4.1 <code>addTrailer</code>	88
10.14.4.2 <code>addTrailers</code>	89
10.14.4.3 <code>end</code>	89
10.14.4.4 <code>finished</code>	89
10.14.4.5 <code>flush()</code>	90
10.14.4.6 <code>headers()</code> [1/2]	90
10.14.4.7 <code>headers()</code> [2/2]	90
10.14.4.8 <code>setOptions()</code>	91
10.14.4.9 <code>write</code>	91
10.14.4.10 <code>writeContinue</code>	92
10.14.4.11 <code>writeHead</code> [1/6]	92
10.14.4.12 <code>writeHead</code> [2/6]	92
10.14.4.13 <code>writeHead</code> [3/6]	93
10.14.4.14 <code>writeHead</code> [4/6]	93
10.14.4.15 <code>writeHead</code> [5/6]	93
10.14.4.16 <code>writeHead</code> [6/6]	93

10.15Tufao::HttpsServer Class Reference	94
10.15.1 Detailed Description	96
10.15.2 Member Function Documentation	96
10.15.2.1 setLocalCertificate()	96
10.15.2.2 setPrivateKey()	97
10.16Tufao::HttpUpgradeRouter Class Reference	97
10.16.1 Detailed Description	99
10.16.2 Member Typedef Documentation	100
10.16.2.1 Handler	100
10.16.3 Member Function Documentation	100
10.16.3.1 handleUpgrade	100
10.16.3.2 map() [1/2]	101
10.16.3.3 map() [2/2]	101
10.17Tufao::IByteArray Class Reference	102
10.17.1 Detailed Description	103
10.18Tufao::HttpUpgradeRouter::Mapping Struct Reference	103
10.18.1 Detailed Description	104
10.18.2 Member Data Documentation	104
10.18.2.1 path	104
10.19Tufao::HttpRequestRouter::Mapping Struct Reference	105
10.19.1 Detailed Description	105
10.19.2 Member Data Documentation	106
10.19.2.1 method	106
10.19.2.2 path	106
10.20Tufao::NotFoundHandler Class Reference	106
10.20.1 Detailed Description	108
10.20.2 Constructor & Destructor Documentation	108
10.20.2.1 NotFoundHandler()	108
10.20.3 Member Function Documentation	108
10.20.3.1 handler()	109

10.21 Tufao::Session::PropertyWrapper Class Reference	109
10.21.1 Detailed Description	110
10.21.2 Constructor & Destructor Documentation	110
10.21.2.1 PropertyWrapper()	110
10.22 Tufao::Session Class Reference	110
10.22.1 Detailed Description	111
10.22.2 Constructor & Destructor Documentation	112
10.22.2.1 Session()	112
10.22.3 Member Function Documentation	112
10.22.3.1 apply() [1/2]	112
10.22.3.2 apply() [2/2]	113
10.22.3.3 hasValue()	114
10.22.3.4 setValue()	114
10.22.3.5 value()	114
10.23 Tufao::SessionSettings Struct Reference	115
10.23.1 Detailed Description	116
10.23.2 Member Data Documentation	116
10.23.2.1 domain	116
10.23.2.2 httpOnly	117
10.23.2.3 name	117
10.23.2.4 path	118
10.23.2.5 secure	118
10.23.2.6 timeout	119
10.24 Tufao::SessionStore Class Reference	119
10.24.1 Detailed Description	122
10.24.2 Constructor & Destructor Documentation	126
10.24.2.1 SessionStore()	126
10.24.3 Member Function Documentation	126
10.24.3.1 defaultSettings()	127
10.24.3.2 hasProperty()	127

10.24.3.3 properties()	127
10.24.3.4 property()	128
10.24.3.5 removeProperty()	128
10.24.3.6 removeSession()	128
10.24.3.7 resetSession()	128
10.24.3.8 session() [1/2]	129
10.24.3.9 session() [2/2]	129
10.24.3.10setMacSecret()	129
10.24.3.11setProperty()	130
10.24.3.12setSession()	130
10.24.3.13unsetSession()	130
10.24.4 Member Data Documentation	131
10.24.4.1 settings	131
10.25Tufao::SimpleSessionStore Class Reference	131
10.25.1 Detailed Description	134
10.25.2 Constructor & Destructor Documentation	134
10.25.2.1 SimpleSessionStore()	135
10.25.3 Member Function Documentation	135
10.25.3.1 defaultInstance()	135
10.25.3.2 refreshInterval()	135
10.26Tufao::UrlRewriterHandler Class Reference	136
10.26.1 Detailed Description	138
10.26.2 Constructor & Destructor Documentation	138
10.26.2.1 UrlRewriterHandler()	138
10.26.3 Member Function Documentation	138
10.26.3.1 handler()	138
10.26.3.2 handleRequest	139
10.26.3.3 setUrl()	139
10.26.3.4 url()	139
10.27Tufao::WebSocket Class Reference	140

10.27.1 Detailed Description	143
10.27.2 Constructor & Destructor Documentation	143
10.27.2.1 WebSocket()	144
10.27.3 Member Function Documentation	144
10.27.3.1 connectToHost() [1/4]	144
10.27.3.2 connectToHost() [2/4]	144
10.27.3.3 connectToHost() [3/4]	145
10.27.3.4 connectToHost() [4/4]	145
10.27.3.5 connectToHostEncrypted() [1/4]	145
10.27.3.6 connectToHostEncrypted() [2/4]	146
10.27.3.7 connectToHostEncrypted() [3/4]	146
10.27.3.8 connectToHostEncrypted() [4/4]	146
10.27.3.9 messagesType()	147
10.27.3.10 peerAddress()	147
10.27.3.11 peerPort()	148
10.27.3.12 ping	148
10.27.3.13 pong	148
10.27.3.14 sendBinaryMessage	149
10.27.3.15 sendUtf8Message	149
10.27.3.16 setMessagesType()	149
10.27.3.17 startServerHandshake() [1/2]	150
10.27.3.18 startServerHandshake() [2/2]	151
11 File Documentation	153
11.1 httpserverplugin.h File Reference	153
11.1.1 Macro Definition Documentation	154
11.1.1.1 TUFAO_HTTPSERVERPLUGIN_IID	154
Index	155

Chapter 1

Main Page

1.1 Introduction

Tufão is a web framework for C++ that makes use of Qt's object communication system (signals & slots). It features:

- High performance standalone server
- Cross-plataform support
- Good documentation
- Support modern HTTP features
 - Persistent streams
 - Chunked entities
 - 100-continue status
 - WebSocket
- HTTPS support
- Flexible request router
- Static file server with support for conditional requests, partial download and automatic mime detection
- Plugin-based server to allow change the running code without restart the application
- Flexible and secure session support
- QtCreator's plugin to allow create new applications rapidly
- Timeout support

It uses Ryan Dahl's HTTP parser to provide good performance.

1.2 Creating a Tufão-based application

In this section, I assume that you already have installed Tufão libraries and tools (QtCreator's plugin, ...). If you have any trouble following the [build/install steps](#), [click here](#).

1.2.1 Hello World!

Let's start by examining a *Hello World* application:

```
00001 #include <QCoreApplication>
00002
00003 #include <Tufao/HttpServer>
00004 #include <Tufao/HttpServerResponse>
00005
00006 using namespace Tufao;
00007
00008 int main(int argc, char *argv[])
00009 {
00010     QCoreApplication a(argc, argv);
00011
00012     HttpServer server;
00013
00014     QObject::connect(&server, &HttpServer::requestReady,
00015                     [] (HttpServerRequest &, HttpServerResponse &res) {
00016                         res.writeHead(200, "OK");
00017                         res.end("Hello World\n");
00018                     });
00019
00020     server.listen(QHostAddress::Any, 8080);
00021
00022     return a.exec();
00023 }
```

The code shown above is what you need to create a program that will respond with Hello World to every request. It depends on QtCore, QtNetwork and Tufão.

In the lines 1-4, we import all definitions we need to this code work and in line 6, we import names from the [Tufao](#) namespace into global namespace.

In the function main, we declare our event loop (line 10) that allow us to handle the server's connections asynchronously and in line 22, we start it.

In line 12, we declare the server object, responsible for manage HTTP messages. We can access the HTTP requests through the requestReady signal, as shown in lines 14 to 18. Then, we tell the server to listen for incoming connections on port 8080 (line 20) and our application is ready to serve HTTP clients.

All you need to handle a HTTP session is the pair of HttpServerRequest and HttpServerResponse objects. The HttpServerRequest exports the input of the session and the HttpServerResponse is the output, from the point of view of your application server.

Every HTTP response is made of three parts:

- A status-line
- Some meta-data with key-value pairs (the headers)
- A body (anything from text to binary-based data)

In every response, you **must** call all these methods, in this order:

1. writeHead* to write the status-line
2. end* to close the HTTP session

The headers are optional and you can set them before do any call to write or – obviously – end. See [Tufao::HttpServerResponse](#) for more details.

The handler used as parameter in QObject::connect function is a C++11 lambda, but you can use whatever Q←Object::connect accepts. In our handler, we just write a simple "Hello World" message.

1.2.2 Revisiting Hello World!

Now that you got warm, let's see an example more complicated.

First of all, you must open QtCreator and go to the new project dialog, you may be able to select Tufão Web Server project from the list of projects in the Tufão category.

Go through the pages of the project wizard, selecting the *application* template. At the end, you'll get an unconfigured qmake-based application. Under QtCreator's interface, go to the *projects mode* and choose a configuration. If you don't configure the project, QtCreator won't be able to compile your project.

1.2.3 And then, there was the code

You may have a *main.cpp* file with a code like this:

```
00001 #include <QCoreApplication>
00002
00003 #include <Tufao/HttpPluginServer>
00004 #include <Tufao/HttpFileServer>
00005 #include <Tufao/NotFoundHandler>
00006
00007 #include <Tufao/HttpServerRequestRouter>
00008 #include <Tufao/HttpServer>
00009
00010 using namespace Tufao;
00011
00012 int main(int argc, char *argv[])
00013 {
00014     QCoreApplication a(argc, argv);
00015
00016     HttpPluginServer plugins("routes.json");
00017
00018     HttpServerRequestRouter router{
00019         {QRegularExpression{""}, plugins},
00020         {QRegularExpression{""}, HttpFileServer::handler("public")},
00021         {QRegularExpression{""}, NotFoundHandler::handler()}
00022     };
00023
00024     HttpServer server;
00025
00026     QObject::connect(&server, &HttpServer::requestReady,
00027                     &router, &HttpServerRequestRouter::handleRequest
00028     );
00029     server.listen(QHostAddress::Any, 8080);
00030
00031     return a.exec();
00032 }
```

A main file with 32 lines of code. It's a bunch of code, but they'll prove their value.

Tufão provides a web framework and has its own HTTP server (based on Ryan Dahl's HTTP parser). This design give you greater control over the applications and can really boost its performance. But, as uncle Ben may have told you:

With great power comes great responsibility

This may sound scary to you at first, but don't worry, the Tufão team will do their best to make this responsibility easier to assume, with each realease. But you need to promisse that you'll use Tufão facilities and read the documentation.

To use the Tufão HTTP server, we instantiate, as show in the line 24, a [Tufao::HttpServer](#), put it to listen on port 8080, as show in the line 29, and start a event loop, as show in the line 31. This server will expose all we need to this introduction.

The [Tufao::HttpServer](#) will emit the [Tufao::HttpServer::requestReady](#) signal each time a request comes. We can handle all requests in a single place in the code, but we won't do that, because... em... as some wise person said before, we need:

Divide and conquer

What I mean is that every request **IS** different and every one needs a different handler. The convention is to serve different content under different paths. This is what we'll do.

We could decompose our slot into some functions to do the job. But we won't do that, because Tufão already has a better abstraction for the problem, [Tufao::HttpServerRequestRouter](#).

The idea behind [Tufao::HttpServerRequestRouter](#) is to glue a chain of request handlers and mapped paths to them. If a request comes to a handler unable to handle it, the request is delegated to another handler in the chain.

In the code, we have one router (line 18) and three handlers (lines 16/19, 20 and 21). In line 26-27, we bind the router and the server. The handlers, in order, are:

- [Tufao::HttpPluginServer](#): Uses plugins to handle the requests. A plugin mechanism is what allows you to change the running code without restart the application. In this code, we use the file routes.json to configure the plugins.
- [Tufao::HttpFileServer](#): Serve static files. We use the folder *public* as root dir.
- [Tufao::NotFoundHandler](#): A handler that responds to every request with a 404 status code.

See [Tufão's plugin system](#) to learn how to create and attach plugins to the running application.

If you access <http://localhost:8080/> you'll see a message telling you that the page wasn't found. Create a folder called *public* in the working dir of the executable and put a file called *index.html* there:

```
<html>
  <head><title>Hello World</title></head>
  <body>
    <h1>Congratulations,</h1>
    <p>you are able to start web development in C++ with Tufão</p>
  </body>
</html>
```

Now access the page <http://localhost:8080/index.html> and see the result. As an exercise, I left the task to add a *favicon* to your application using the [Tufão's plugin system](#).

1.2.4 Build system integration

The Tufão has native support to three build systems: qmake, pkgconfig and CMake.

Once Tufão is correctly installed, you can use Tufão in your qmake-based applications appending the following line to your project file, where *x* is the major version (0 for Tufão 0.x and 1 for Tufão 1.x) you want to use:

```
CONFIG += TUFAOx
```

If you use CMake, just follow the common steps (require package and handle compile and link flags) in your CMakeLists.txt file:

```
find_package(Tufao1 1.0 REQUIRED)
include_directories("${TUFAO_INCLUDE_DIR}")
target_link_libraries(foofoo ${TUFAO_LIBRARIES})
```

If you want to use a FindTufao.cmake file embedded in your project dir, add the following line to your CMakeLists.txt:

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "/path/to/FindTufao.cmake")
```

If you want to use autotools or another build system that has pkgconfig support, you can use the Tufão's pkgconfig module, where *x* is the major version (0 for Tufão 0.x and 1 for Tufão 1.x) you want to use:

```
$ pkg-config tufaox
```

If want to use a build system that don't support pkgconfig, you should be ashamed, but you may still be able to use Tufão (probably editing the compiler -I flag and the linker -L and -l flags).

Chapter 2

Design Model

This page describes Tufão's design model.

Just like Node.JS, Tufão uses an async event-driven model. That is, Tufão will handle multiple connections per thread. The Qt executor (i.e. `QCoreApplication`, `QThread` or other) will take care of the scheduling tasks required to handle multiple connections per thread. You can also spawn multiple executors in different threads and delegate different handlers to different executors, then you'll have a multiple-connections-per-event-loop/one-event-loop-per-thread architecture.

The server will begin to work once the control goes to the Qt executor (e.g. `QCoreApplication::exec`).

The simplest choice is just to use single-threaded async I/O.

Chapter 3

Tufão's plugin system

When you are developing web applications, usually you are faced with the problem of develop applications that will be running for years, without stop.

Solutions in interpreted languages usually handle this problem by reloading the source file each time it changes. In C++, we can achieve a similar behaviour through plugins.

Tufão provides the class [Tufao::HttpPluginServer](#) and a QtCreator's plugin to facilitate the integration of Qt plugins and [Tufao::HttpServerRequestRouter](#).

3.1 Understanding by example

To help you understand the Tufão's plugin system, let's revisit the application created in [Revisiting Hello World!](#). This application should be the same generated by the *application* template, under Tufão QtCreator's plugin.

3.1.1 The Tufao::HttpPluginServer class

The [Tufao::HttpServerRequestRouter](#) provides a robust request router interface and would be useful if we could tell it to use a plugin located in the file system as a handler. This is where [Tufao::HttpPluginServer](#) enters.

[Tufao::HttpPluginServer](#) implements the [Tufao::AbstractHttpServerRequestHandler](#) interface, then you can use it as a handler in [Tufao::HttpServerRequestRouter](#). [Tufao::HttpPluginServer](#) has its own set of mapping rules and handlers, but it will load its handlers from plugins.

In the plugin server created previously, we pass a file in the constructor. This is how we set the plugins from [Tufao::HttpPluginServer](#). This file can be edited using any text editor and Tufão will reload it when the file changes. See [Tufao::HttpPluginServer::setConfig](#) to learn how edit this file.

3.1.2 The plugin

Enough text! Let's create our plugin. First, you must open QtCreator, go to the new project dialog and select the *plugin* template in the *Tufão Web Server project*.

You'll get the following files:

- *.pro: The name of this file depends on the name of the project. Tells qmake to use the *lib TEMPLATE* and the *plugin CONFIG*.
- plugin.h: Defines a class that implements the [Tufao::HttpServerPlugin](#) interface, needed by [Tufao::HttpPluginServer](#).
- plugin.cpp: Contains the implementation of Plugin members.

The [Tufao::HttpServerPlugin](#) class is just a *factory* used by [Tufao::HttpPluginServer](#) to instantiate new handler objects. If you intend to create a plugin to be used by [Tufao::HttpPluginServer](#), you must:

1. Use Qt's plugin system
 - (a) Create a new qmake-based project
 - (b) Use *lib* template and *plugin* config
2. Implement the [Tufao::HttpServerPlugin](#) interface in the plugin class (and don't forget the *Q_INTERFACES* macro)
3. Use *Q_PLUGIN_METADATA* to tell Qt about your plugin's metadata

See [Tufao::HttpServerPlugin](#) for more details.

3.1.3 The glue!

Finally we have:

- An application supporting plugins
- A plugin

The only thing missing here is tell the application to use the plugin we created. To do that, we need:

- Edit the config file used by the plugin server of our application. In the previous application, this file is any file named *routes.json* placed in the application's working dir.

Here is a basic one (adapt and use it):

```
{
  "version": 0,
  "plugins": [
    {
      "name": "myplugin",
      "path": "/path/to/your/plugin/binary"
    }
  ],
  "requests": [
    {
      "path": "^/plugin$",
      "plugin": "myplugin"
    }
  ]
}
```

When you ran your application, Tufão tried to find the *routes.json* file and failed. Now you'll restart the application and the application will succeed to find the *routes.json* file.

After Tufão finds the *routes.json* file, it will watch this file for changes and reload the plugins when it happens.

See also

[Tufao::HttpServerPlugin](#)

Chapter 4

Safe signals

We say that a signal is safe when it's safe to delete the object in the slot connected to this signal.

This property implies that no member-function will access the attributes of the object after emit the signal (and won't schedule such accesses).

Consider the following code:

```
void SomeObject::someMethod()
{
    emit someSignal();
    qDebug() << this->x;
}
```

In the previous code, *someSignal* is an unsafe signal, because there is accesses to the internal state of the object after it emits the signal. We can refactor the previous code to make *someSignal* safe by reordering. See the final version, where *someSignal* is safe:

```
void SomeObject::someMethod()
{
    qDebug() << this->x;
    emit someSignal();
}
```

Why do you wanna a safe signal? They are **safer** and you need to worry less about invalid access. Consider you have an object *O* – allocated on the heap – of type *SomeObject* and a method *M* connected to the signal *someSignal*, then the following code is valid:

```
class OurClass: public QObject
{
    Q_OBJECT
public:
    OurClass(SomeObject *O, QObject *parent) :
        QObject(parent), O(O)
    {
        connect(&O, &SomeObject::someSignal, this, &OurClass::M);
    }

    void M()
    {
        delete O;
    }

private:
    SomeObject *O;
}
```

If you have an unsafe signal, your only choice is to use `QObject::deleteLater`.

4.1 Why some signals are unsafe

Sometimes, it's too complicated to create a safe signal. Consider you receive a message via the network and some method parse this message. Consider the method should emit signals each time a new message is ready, something like the code below:

```
void Foobar::slot()
{
    QByteArray buf = socket->readAll();
    while (buf.size()) {
        // ...

        if (/* ... */)
            emit unsafeSignal();

        // ...
    }
}
```

The above code uses an unsafe signal. Sure you could convert the code to use safe signals, but the overhead of performance and responsiveness created wouldn't be worth.

Another example of an unsafe signal:

```
void Foobar::method()
{
    // ...

    emit conditionOneReady();

    // ...

    emit conditionTwoReady();

    // ...

    // The below signal is the only safe signal
    emit finalConditionReady();
}
```

4.2 Conclusion

Every signal in Tufão is safe, unless explicitly stated the opposite.

Warning

It's unsafe to *delete* an object in the body of a slot connected to an unsafe signal emitted by the same object.

Some alternative ways of *deleting* an object in code triggered by an unsafe signal are:

- Call `QObject::deleteLater`
- Use a Qt's `Qt::QueuedConnection` connection to make the connection between the unsafe signal and your slot
- Queuing/delegating the responsibility to somebody else
- Use some well designed architecture on top of `QSharedPointer` or similar

Chapter 5

Namespace Index

5.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Tufao	This is the namespace where all Tufão facilities are grouped	19
-----------------------	--	--------------------

Chapter 6

Hierarchical Index

6.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Tufao::AbstractHttpRequestHandler	25
Tufao::ClassHandlerManager	38
Tufao::HttpFileServer	45
Tufao::HttpPluginServer	51
Tufao::HttpRequestRouter	78
Tufao::NotFoundHandler	106
Tufao::UrlRewriterHandler	136
Tufao::AbstractHttpUpgradeHandler	27
Tufao::HttpUpgradeRouter	97
Tufao::ClassHandlerPluginInfo	41
Tufao::HttpServerPlugin	65
Tufao::HttpUpgradeRouter::Mapping	103
Tufao::HttpRequestRouter::Mapping	105
Tufao::Session::PropertyWrapper	109
QByteArray	
Tufao::IByteArray	102
QMultiHash	
Tufao::Headers	42
QObject	
Tufao::AbstractMessageSocket	31
Tufao::WebSocket	140
Tufao::ClassHandler	35
Tufao::ClassHandlerManager	38
Tufao::HttpFileServer	45
Tufao::HttpPluginServer	51
Tufao::HttpServer	57
Tufao::HttpsServer	94
Tufao::HttpRequest	67
Tufao::HttpRequestRouter	78
Tufao::HttpResponse	84
Tufao::HttpUpgradeRouter	97
Tufao::NotFoundHandler	106
Tufao::SessionStore	119
Tufao::SimpleSessionStore	131
Tufao::UrlRewriterHandler	136
Tufao::Session	110
Tufao::SessionSettings	115

Chapter 7

Class Index

7.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Tufao::AbstractHttpRequestHandler	25
This class provides an interface for HttpRequest handlers	
Tufao::AbstractHttpUpgradeHandler	27
This class provides an interface for HTTP Upgrade handlers	
Tufao::AbstractMessageSocket	31
The Tufao::AbstractMessageSocket class represents a socket that sends and receives messages	
Tufao::ClassHandler	35
The ClassHandler class Define the interface to implement when creating a plugin	
Tufao::ClassHandlerManager	38
This class serves as the HttpRequestHandler for all ClassHandler plugins	
Tufao::ClassHandlerPluginInfo	41
Tufao::Headers	42
This class provides a representation of HTTP headers	
Tufao::HttpFileServer	45
You can use this class to serve static files under Tufão	
Tufao::HttpPluginServer	51
This class provides a plugin-based request handler	
Tufao::HttpServer	57
The Tufao::HttpServer class provides an implementation of the HTTP protocol	
Tufao::HttpServerPlugin	65
This class provides a factory interface to create request handlers and communicate with factories of other plugins	
Tufao::HttpRequest	67
The Tufao::HttpServer represents a HTTP request received by Tufao::HttpServer	
Tufao::HttpRequestRouter	78
This class provides a robust and high performance HTTP request router	
Tufao::HttpResponse	84
The Tufao::HttpResponse is used to respond to a Tufao::HttpRequest	
Tufao::HttpsServer	94
Tufao::HttpsServer is a subclass of Tufao::HttpServer that provides secure communication with the client	
Tufao::HttpUpgradeRouter	97
This class provides a robust and high performance HTTP request router	
Tufao::IByteArray	102
This class provides a case insensitive QByteArray	

Tufao::HttpUpgradeRouter::Mapping	
This class describes a request handler and a filter	103
Tufao::HttpServerRequestRouter::Mapping	
This class describes a request handler and a filter	105
Tufao::NotFoundHandler	
A handler that responds with "Not found" to every request	106
Tufao::Session::PropertyWrapper	
Provides a object that give less verbose access to a session property	109
Tufao::Session	
This class provides easier access to the session's properties	110
Tufao::SessionSettings	
Exposes details that sessions use to handle cookies	115
Tufao::SessionStore	
SessionStore class can be used to store data that must persist among different requests	119
Tufao::SimpleSessionStore	
SimpleSessionStore implements a simple storage mechanism to SessionStore	131
Tufao::UrlRewriterHandler	
This class provides a handler to internally (only seen by your application) rewrite the URL	136
Tufao::WebSocket	
This class represents a WebSocket connection	140

Chapter 8

File Index

8.1 File List

Here is a list of all documented files with brief descriptions:

abstracthttpserverrequesthandler.h	??
abstracthttpupgradehandler.h	??
abstractmessagesocket.h	??
classhandler.h	??
classhandlermanager.h	??
design-model.h	??
headers.h	??
httpfileserver.h	??
httppluginserver.h	??
httpserver.h	??
httpserverplugin.h	153
httpserverrequest.h	??
httpserverrequestrouter.h	??
httpserverresponse.h	??
httpsserver.h	??
httpupgraderouter.h	??
ibytearray.h	??
main.h	??
notfoundhandler.h	??
pluginsystem.h	??
safe-signal.h	??
session.h	??
sessionsettings.h	??
sessionstore.h	??
simplesessionstore.h	??
tufao_global.h	??
urlrewriterhandler.h	??
websocket.h	??

Chapter 9

Namespace Documentation

9.1 Tufao Namespace Reference

This is the namespace where all Tufão facilities are grouped.

Classes

- class [AbstractHttpRequestHandler](#)
This class provides an interface for [HttpRequest](#) handlers.
- class [AbstractHttpUpgradeHandler](#)
This class provides an interface for HTTP Upgrade handlers.
- class [AbstractMessageSocket](#)
The [Tufao::AbstractMessageSocket](#) class represents a socket that sends and receives messages.
- class [ClassHandler](#)
The [ClassHandler](#) class Define the interface to implement when creating a plugin.
- class [ClassHandlerManager](#)
This class serves as the [HttpRequestHandler](#) for all [ClassHandler](#) plugins.
- struct [ClassHandlerPluginInfo](#)
- struct [Headers](#)
This class provides a representation of HTTP headers.
- class [HttpFileServer](#)
You can use this class to serve static files under Tufão.
- class [HttpPluginServer](#)
This class provides a plugin-based request handler.
- class [HttpServer](#)
The [Tufao::HttpServer](#) class provides an implementation of the HTTP protocol.
- class [HttpServerPlugin](#)
This class provides a factory interface to create request handlers and communicate with factories of other plugins.
- class [HttpRequest](#)
The [Tufao::HttpServer](#) represents a HTTP request received by [Tufao::HttpServer](#).
- class [HttpRequestRouter](#)
This class provides a robust and high performance HTTP request router.
- class [HttpServerResponse](#)
The [Tufao::HttpServerResponse](#) is used to respond to a [Tufao::HttpRequest](#).
- class [HttpsServer](#)

[Tufao::HttpsServer](#) is a subclass of [Tufao::HttpServer](#) that provides secure communication with the client.

- class [HttpUpgradeRouter](#)

This class provides a robust and high performance HTTP request router.

- class [IByteArray](#)

This class provides a case insensitive QByteArray.

- class [NotFoundHandler](#)

A handler that responds with "Not found" to every request.

- class [Session](#)

This class provides easier access to the session's properties.

- struct [SessionSettings](#)

The [SessionSettings](#) class exposes details that sessions use to handle cookies.

- class [SessionStore](#)

[SessionStore](#) class can be used to store data that must persist among different requests.

- class [SimpleSessionStore](#)

[SimpleSessionStore](#) implements a simple storage mechanism to [SessionStore](#).

- class [UrlRewriterHandler](#)

This class provides a handler to internally (only seen by your application) rewrite the URL.

- class [WebSocket](#)

This class represents a [WebSocket](#) connection.

Enumerations

- enum [HttpVersion](#) { HTTP_1_0, HTTP_1_1 }

- enum [HttpResponseStatus](#) {
CONTINUE = 100, **SWITCHING_PROTOCOLS** = 101, **PROCESSING** = 102, **CHECKPOINT** = 103,
OK = 200, **CREATED** = 201, **ACCEPTED** = 202, **NON_AUTHORITATIVE_INFORMATION** = 203,
NO_CONTENT = 204, **RESET_CONTENT** = 205, **PARTIAL_CONTENT** = 206, **MULTI_STATUS** = 207,
ALREADY_REPORTED = 208, **IM_USED** = 226, **MULTIPLE_CHOICES** = 300, **MOVED_PERMANENTLY**
= 301,
FOUND = 302, **SEE_OTHER** = 303, **NOT_MODIFIED** = 304, **USE_PROXY** = 305,
SWITCH_PROXY = 306, **TEMPORARY_REDIRECT** = 307, **RESUME_INCOMPLETE** = 308, **BAD_REQUEST** = 400,
UNAUTHORIZED = 401, **PAYMENT_REQUIRED** = 402, **FORBIDDEN** = 403, **NOT_FOUND** = 404,
METHOD_NOT_ALLOWED = 405, **NOT_ACCEPTABLE** = 406, **PROXY_AUTHENTICATION_REQUIRED**
= 407, **REQUEST_TIMEOUT** = 408,
CONFLICT = 409, **GONE** = 410, **LENGTH_REQUIRED** = 411, **PRECONDITION_FAILED** = 412,
REQUEST_ENTITY_TOO_LARGE = 413, **REQUEST_URI_TOO_LONG** = 414, **UNSUPPORTED_MEDIA_TYPE** = 415,
REQUESTED_RANGE_NOT_SATISFIABLE = 416,
EXPECTATION_FAILED = 417, **I_AM_A_TEAPOT** = 418, **UNPROCESSABLE_ENTITY** = 422, **LOCKED** =
423,
FAILED_DEPENDENCY = 424, **UNORDERED_COLLECTION** = 425, **UPGRADE_REQUIRED** = 426, **PRECONDITION_REQUIRED** = 428,
TOO_MANY_REQUESTS = 429, **REQUEST_HEADER_FIELDS_TOO_LARGE** = 431, **NO_RESPONSE** =
444, **RETRY_WITH** = 449,
CLIENT_CLOSED_REQUEST = 499, **INTERNAL_SERVER_ERROR** = 500, **NOT_IMPLEMENTED** = 501,
BAD_GATEWAY = 502,
SERVICE_UNAVAILABLE = 503, **GATEWAY_TIMEOUT** = 504, **HTTP_VERSION_NOT_SUPPORTED** =
505, **VARIANT_ALSO_NEGOTIATES** = 506,
INSUFFICIENT_STORAGE = 507, **LOOP_DETECTED** = 508, **BANDWIDTH_LIMIT_EXCEEDED** = 509,
NOT_EXTENDED = 510 }

The values in this enum represents a HTTP status code.

- enum [WebSocketError](#) {
[WebSocketError::NO_ERROR](#) = 0, [WebSocketError::CONNECTION_REFUSED](#), [WebSocketError::REMOTE_HOST_CLOSED](#),
[WebSocketError::HOST_NOT_FOUND](#),
[WebSocketError::ACCESS_ERROR](#), [WebSocketError::OUT_OF_RESOURCES](#), [WebSocketError::SOCKET_TIMEOUT](#),
[WebSocketError::NETWORK_ERROR](#),
[WebSocketError::UNSUPPORTED_SOCKET_OPERATION](#), [WebSocketError::PROXY_AUTHENTICATION_REQUIRED](#),
[WebSocketError::SSL_HANDSHAKE_FAILED](#), [WebSocketError::PROXY_CONNECTION_REFUSED](#),
[WebSocketError::PROXY_CONNECTION_CLOSED](#), [WebSocketError::PROXY_CONNECTION_TIMEOUT](#),
[WebSocketError::PROXY_NOT_FOUND](#), [WebSocketError::PROXY_PROTOCOL_ERROR](#),
[WebSocketError::WEBSOCKET_HANDSHAKE_FAILED](#), [WebSocketError::WEBSOCKET_PROTOCOL_ERROR](#),
[WebSocketError::UNKNOWN_ERROR](#) }
This enum describes the possible errors that can occur.
- enum [WebSocketMessageType](#) { [WebSocketMessageType::TEXT_MESSAGE](#), [WebSocketMessageType::BINARY_MESSAGE](#) }
This enum represents the possible message's types that WebSocket supports.

Functions

- TUFao_EXPORT QDebug **operator**<< (QDebug dbg, const [Headers](#) &headers)
- [HttpServerResponse](#) & **operator**<< ([HttpServerResponse](#) &response, const QByteArray &chunk)
This overload allows you to use a [HttpServerResponse](#) object as a stream.
- bool **operator**!= (const [IByteArray](#) &lhs, const [IByteArray](#) &rhs)
- bool **operator**< (const [IByteArray](#) &lhs, const [IByteArray](#) &rhs)
- bool **operator**<= (const [IByteArray](#) &lhs, const [IByteArray](#) &rhs)
- bool **operator**== (const [IByteArray](#) &lhs, const [IByteArray](#) &rhs)
- bool **operator**> (const [IByteArray](#) &lhs, const [IByteArray](#) &rhs)
- bool **operator**>= (const [IByteArray](#) &lhs, const [IByteArray](#) &rhs)
- uint **qHash** (const [IByteArray](#) &key)

9.1.1 Detailed Description

This is the namespace where all Tufão facilities are grouped.

Its purpose is isolate its symbols from symbols of other packages, avoiding collision problems.

9.1.2 Enumeration Type Documentation

9.1.2.1 [HttpResponseStatus](#)

```
enum Tufao::HttpResponseStatus [strong]
```

The values in this enum represents a HTTP status code.

These are sent in the first line of a HTTP response message. You should consult external doc (rfc 2616) to know when to use each value. The HTTP status code most used is OK.

You can use the values in this enum in `Tufao::HttpServerResponse::writeHead(int)`.

Since

1.0

9.1.2.2 WebSocketError

```
enum Tufao::WebSocketError [strong]
```

This enum describes the possible errors that can occur.

Note

Avoid to directly test against NO_ERROR, because Windows API defines the NO_ERROR macro and your code might fail to build under this platform. You can safely make a test using code like the following:

```
if (int(ws.error()))
    /* ... */;
```

Since

1.0

Enumerator

NO_ERROR	No error.
CONNECTION_REFUSED	See QAbstractSocket::ConnectionRefusedError. It can happen during the opening handshake (only when acting as client).
REMOTE_HOST_CLOSED	See QAbstractSocket::RemoteHostClosedError. It can happen during the opening handshake (only when acting as client).
HOST_NOT_FOUND	See QAbstractSocket::HostNotFoundError. It can happen during the opening handshake (only when acting as client).
ACCESS_ERROR	See QAbstractSocket::SocketAccessError. It can happen during the opening handshake (only when acting as client).
OUT_OF_RESOURCES	See QAbstractSocket::SocketResourceError. It can happen during the opening handshake (only when acting as client).
SOCKET_TIMEOUT	See QAbstractSocket::SocketTimeoutError. It can happen during the opening handshake (only when acting as client).
NETWORK_ERROR	See QAbstractSocket::NetworkError. It can happen during the opening handshake (only when acting as client).
UNSUPPORTED_SOCKET_OPERATION	See QAbstractSocket::UnsupportedSocketOperationError. It can happen during the opening handshake (only when acting as client).
PROXY_AUTHENTICATION_REQUIRED	See QAbstractSocket::ProxyAuthenticationRequiredError. It can happen during the opening handshake (only when acting as client).
SSL_HANDSHAKE_FAILED	See QAbstractSocket::SslHandshakeFailedError. It can happen during the opening handshake (only when acting as client).
PROXY_CONNECTION_REFUSED	See QAbstractSocket::ProxyConnectionRefusedError. It can happen during the opening handshake (only when acting as client).
PROXY_CONNECTION_CLOSED	See QAbstractSocket::ProxyConnectionClosedError. It can happen during the opening handshake (only when acting as client).
PROXY_CONNECTION_TIMEOUT	See QAbstractSocket::ProxyConnectionTimeoutError. It can happen during the opening handshake (only when acting as client).
PROXY_NOT_FOUND	See QAbstractSocket::ProxyNotFoundError. It can happen during the opening handshake (only when acting as client).

Enumerator

PROXY_PROTOCOL_ERROR	See QAbstractSocket::ProxyProtocolError. It can happen during the opening handshake (only when acting as client).
WEBSOCKET_HANDSHAKE_FAILED	It occurs when the server doesn't support WebSocket for the resource asked for (or for any resource at all). It can happen during the opening handshake (only when acting as client).
WEBSOCKET_PROTOCOL_ERROR	It occurs when the remote peer (or an intermediary) violates the WebSocket protocol.
UNKNOWN_ERROR	Unknown error. You found the chaos.

9.1.2.3 WebSocketMessageType

```
enum Tufao::WebSocketMessageType [strong]
```

This enum represents the possible message's types that [WebSocket](#) supports.

Since

1.0

Enumerator

TEXT_MESSAGE	UTF8 messages.
BINARY_MESSAGE	Binary messages.

9.1.3 Function Documentation

9.1.3.1 operator<<()

```
HttpServerResponse& Tufao::operator<< (
    HttpServerResponse & response,
    const QByteArray & chunk ) [inline]
```

This overload allows you to use a [HttpServerResponse](#) object as a stream.

Note

You still need to call [HttpServerResponse::end](#) when done with the connection and [HttpServerResponse::writeHead](#) before use this operator.

Since

1.0

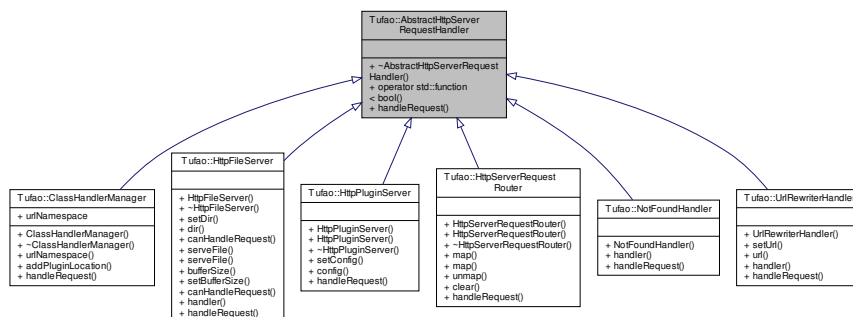
Chapter 10

Class Documentation

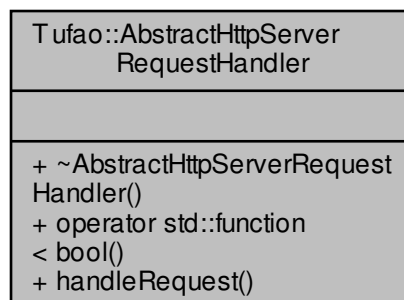
10.1 Tufao::AbstractHttpServerRequestHandler Class Reference

This class provides an interface for [HttpServerRequest](#) handlers.

Inheritance diagram for Tufao::AbstractHttpServerRequestHandler:



Collaboration diagram for Tufao::AbstractHttpServerRequestHandler:



Public Member Functions

- [operator std::function< bool \(HttpServerRequest &, HttpServerResponse &\)>\(\)](#)
Implicit conversion operator to std::function functor object.
- virtual bool [handleRequest](#) ([Tufao::HttpServerRequest](#) &request, [Tufao::HttpServerResponse](#) &response)=0
Handles the `request` using the `response` object.

10.1.1 Detailed Description

This class provides an interface for [HttpServerRequest](#) handlers.

A request handler is usually registered to handle requests matching some set of rules and usually used with a set of other handlers.

A sample request handler is given below:

```
bool RequestHandler::handleRequest(Tufao::HttpServerRequest &request,
                                   Tufao::HttpServerResponse &response)
{
    response.writeHead(HttpStatusCode::OK);
    response.end("Hello World\n");
    return true;
}
```

See also

[HttpServerRequestRouter](#)

Since

0.3

10.1.2 Member Function Documentation

10.1.2.1 [handleRequest\(\)](#)

```
virtual bool Tufao::AbstractHttpServerRequestHandler::handleRequest (
    Tufao::HttpServerRequest & request,
    Tufao::HttpServerResponse & response ) [pure virtual]
```

Handles the `request` using the `response` object.

Return values

<i>true</i>	If the handler has responded to the request.
<i>false</i>	If the requested page can't be generated by this handler. The connection should remain open and the <code>response</code> object shouldn't be used, leaving the response free to be used by other handlers in the chain.

Since

1.0

10.1.2.2 operator std::function< bool()

```
Tufao::AbstractHttpServerRequestHandler::operator std::function< bool (
    HttpServerRequest & ,
    HttpServerResponse & ) [inline]
```

Implicit conversion operator to std::function functor object.

Warning

You shall not use the returned object after the AbstractHttpServerRequest object is destroyed.

Since

1.0

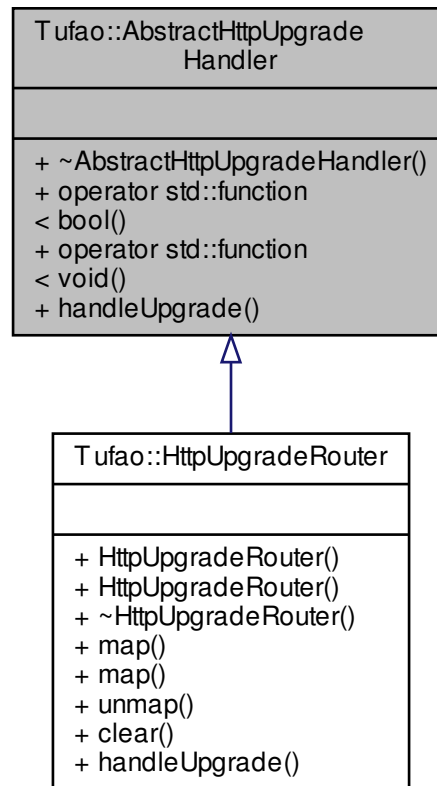
The documentation for this class was generated from the following file:

- abstracthttpserverrequesthandler.h

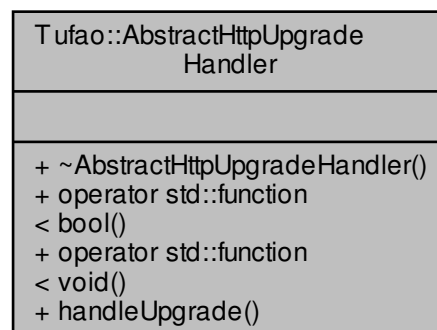
10.2 Tufao::AbstractHttpUpgradeHandler Class Reference

This class provides an interface for HTTP Upgrade handlers.

Inheritance diagram for Tufao::AbstractHttpUpgradeHandler:



Collaboration diagram for Tufao::AbstractHttpUpgradeHandler:



Public Member Functions

- [operator std::function< bool \(HttpServerRequest &, const QByteArray &\)>\(\)](#)
Implicit conversion operator to std::function functor object.
- [operator std::function< void \(HttpServerRequest &, const QByteArray &\)>\(\)](#)
Implicit conversion operator to std::function functor object.
- virtual bool [handleUpgrade](#) ([Tufao::HttpServerRequest](#) &request, const QByteArray &head)=0
Handles the HTTP request.

10.2.1 Detailed Description

This class provides an interface for HTTP Upgrade handlers.

An upgrade handler is usually registered to handle requests matching some set of rules and usually used with a set of other handlers.

A sample upgrade handler is given below:

```
bool RequestHandler::handleUpgrade(Tufao::HttpServerRequest &request,
                                   const QByteArray &head)
{
    Tufao::WebSocket *socket = new Tufao::WebSocket(this);
    socket->startServerHandshake(request, head);
    socket->setMessageType(Tufao::WebSocket::TEXT_MESSAGE);

    connect(socket, SIGNAL(disconnected()), socket, SLOT(deleteLater()));

    connect(socket, SIGNAL(newMessage(QByteArray)),
            this, SIGNAL(newMessage(QByteArray)));
    connect(this, SIGNAL(newMessage(QByteArray)),
            socket, SLOT(sendMessage(QByteArray)));

    return true;
}
```

See also

[HttpUpgradeRouter](#)

Since

0.6

10.2.2 Member Function Documentation

10.2.2.1 handleUpgrade()

```
virtual bool Tufao::AbstractHttpUpgradeHandler::handleUpgrade (
    Tufao::HttpServerRequest & request,
    const QByteArray & head ) [pure virtual]
```

Handles the HTTP request.

Return values

<i>true</i>	If the handler has upgraded the connection to the request protocol.
<i>false</i>	If the handler didn't change the protocol to the requested one. The HTTP session should remain open (eg. a response message shouldn't be sent), leaving the response free to be used by other handlers in the chain.

Since

1.0

10.2.2.2 `operator std::function< bool()`

```
Tufao::AbstractHttpUpgradeHandler::operator std::function< bool (
    HttpRequest & ,
    const QByteArray & ) [inline]
```

Implicit conversion operator to `std::function` functor object.

Warning

You shall not use the returned object after the [AbstractHttpUpgradeHandler](#) object is destroyed.

Since

1.0

10.2.2.3 `operator std::function< void()`

```
Tufao::AbstractHttpUpgradeHandler::operator std::function< void (
    HttpRequest & ,
    const QByteArray & ) [inline]
```

Implicit conversion operator to `std::function` functor object.

Note

The returned functor object calls `handleUpgrade`. If `handleUpgrade` returns false, then it will close the connection.

Warning

You shall not use the returned object after the [AbstractHttpUpgradeHandler](#) object is destroyed.

Since

1.0

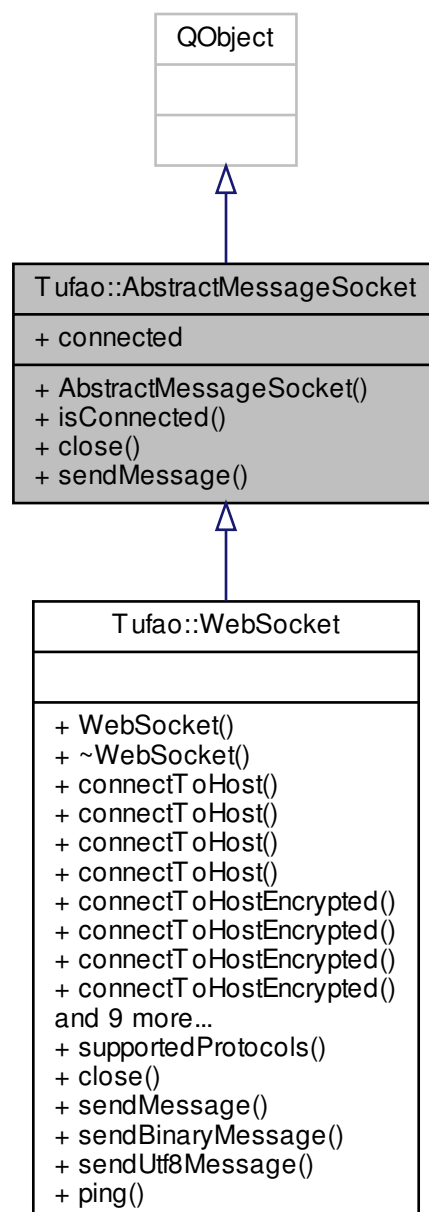
The documentation for this class was generated from the following file:

- `abstracthttpupgradehandler.h`

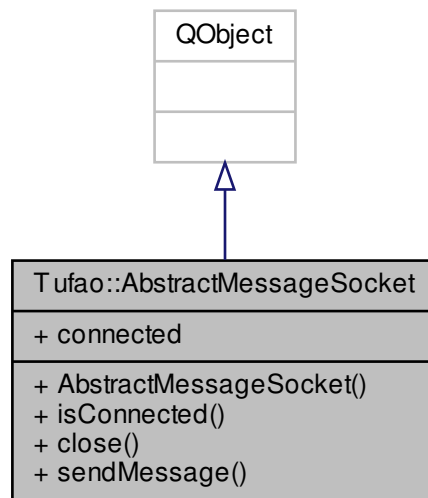
10.3 Tufao::AbstractMessageSocket Class Reference

The [Tufao::AbstractMessageSocket](#) class represents a socket that sends and receives messages.

Inheritance diagram for Tufao::AbstractMessageSocket:



Collaboration diagram for Tufao::AbstractMessageSocket:



Public Slots

- virtual void `close` ()=0
This method should close the connection.
- virtual bool `sendMessage` (const QByteArray &msg)=0
This method should send a new message if the connection is open.

Signals

- void `connected` ()
This signal should be emitted when the connection is open.
- void `disconnected` ()
This signal should be emitted when the connection is closed or when fails to connect.
- void `newMessage` (QByteArray msg)
This signal should be emitted each time a new message is available.

Public Member Functions

- `AbstractMessageSocket` (QObject *parent=0)
Constructs a `Tufao::AbstractMessageSocket` object.
- bool `isConnected` () const
Returns true if the connection is open.

Properties

- bool `connected`

10.3.1 Detailed Description

The [Tufao::AbstractMessageSocket](#) class represents a socket that sends and receives messages.

Classes implementing [Tufao::AbstractMessageSocket](#) can provide several high level functionalities, such as:

- Messages separation and interleaving
- Messages compression
- Authentication
- Proxy or tunnel
- ...

It's main purpose is to allow easily change the message exchange mechanism in algorithms that depends on message passing. You could use it, for example, to allow a class that provide a RPC mechanism to work on different connections types.

Since

0.2

10.3.2 Constructor & Destructor Documentation

10.3.2.1 AbstractMessageSocket()

```
Tufao::AbstractMessageSocket::AbstractMessageSocket (
    QObject * parent = 0 ) [explicit]
```

Constructs a [Tufao::AbstractMessageSocket](#) object.

`parent` is passed to the `QObject` constructor.

10.3.3 Member Function Documentation

10.3.3.1 connected

```
void Tufao::AbstractMessageSocket::connected ( ) [signal]
```

This signal should be emitted when the connection is open.

Note

This signal might be unsafe (read this: [Safe signals](#))!

Unless you know what subclass of [AbstractMessageSocket](#) is being used **AND** this subclass explicitly documents the opposite (signal is safe), you should assume that this signal is unsafe.

10.3.3.2 disconnected

```
void Tufao::AbstractMessageSocket::disconnected ( ) [signal]
```

This signal should be emitted when the connection is closed or when fails to connect.

Note

This signal might be unsafe (read this: [Safe signals](#))!

Unless you know what subclass of [AbstractMessageSocket](#) is being used **AND** this subclass explicitly documents the opposite (signal is safe), you should assume that this signal is unsafe.

10.3.3.3 newMessage

```
void Tufao::AbstractMessageSocket::newMessage (
    QByteArray msg ) [signal]
```

This signal should be emitted each time a new message is available.

Note

This signal might be unsafe (read this: [Safe signals](#))!

Unless you know what subclass of [AbstractMessageSocket](#) is being used **AND** this subclass explicitly documents the opposite (signal is safe), you should assume that this signal is unsafe.

10.3.3.4 sendMessage

```
virtual bool Tufao::AbstractMessageSocket::sendMessage (
    const QByteArray & msg ) [pure virtual], [slot]
```

This method should send a new message if the connection is open.

The object should discard the message if the connection is closed, but it may implement a different behavior (and return true).

Returns

true if successful

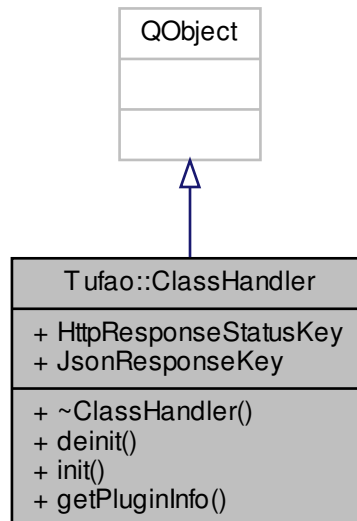
The documentation for this class was generated from the following file:

- abstractmessagesocket.h

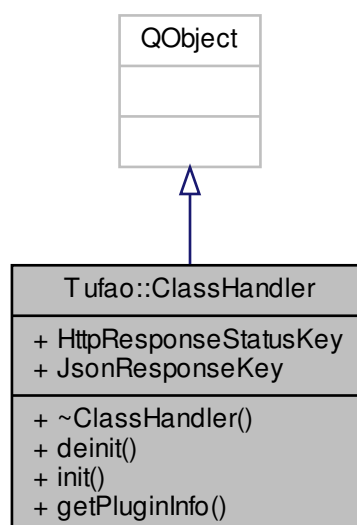
10.4 Tufao::ClassHandler Class Reference

The [ClassHandler](#) class Define the interface to implement when creating a plugin.

Inheritance diagram for Tufao::ClassHandler:



Collaboration diagram for Tufao::ClassHandler:



Public Member Functions

- virtual void [deinit](#) ()=0
Clean up resource; called automatically before plugin is unloaded.
- virtual void [init](#) ()=0
Initialize an instance for use; alled automatically when plugin is registered.
- virtual [ClassHandlerPluginInfo](#) [getPluginInfo](#) () const =0
Get information about the plugin.

Static Public Attributes

- static const QString **HttpResponseStatusKey**
- static const QString **JsonResponseKey**

10.4.1 Detailed Description

The [ClassHandler](#) class Define the interface to implement when creating a plugin.

This interface is used by the [QPluginLoader](#) to load plugins dynamically.

This class is intended to be used in a very specific way. If it is used in that way, and linked into a library, then [Tufao](#) can automatically dispatch incoming requests to this class, and more specifically, to specific methods defined in this class. This allows the developer to just write a class, and not need to worry about crazy regular expressions, and then logic to determine what should be done (action to perform) with the traditional [AbstractHttpRequestHandler](#). As the [AbstractHttpRequestHandler](#) can only have one dispatch-able method, it would have been up to the developer to determine what logic to execute based on the request, or to have a large number of classes & a large number of regular expressions to map to those classes.

This is achieved by breaking the URL paths up into a parameter list, and then invoking a method on the subclass and passing the parameters to that method. So a URL request like: `/forum/read/forumName/someForumName/threadName/someThreadName/page/6` would call a method `Forum::read(QString forumName, QString threadName, int pageNumber)` as `read("someForumName", "someThreadName", 6)`. (please read below for a more accurate description). This would take a user to a forum named *someForumName*, to a topic named *someThreadName*, to the 6th page of responses.

This format is intended to follow the general ReST tenants of resource location, and map them to a method implemented on a [ClassHandler](#). In this example, some resource named *forum* has a sub-resource named *forumName*, and that sub-resource further has a sub-resource named *threadName*, and finally, a further sub-resource of *page*. Each of the first two resources are found by a string, and the last resource is found by an int.

To use this class, there are a few conventions you must follow, and the class needs to be compiled into a plugin. First I'll talk about the implementation, and then I'll talk about the packaging.

Implementation

Creating a Class to match a path

All subclasses must be `Q_OBJECT`'s. Further, two additional macro's must be used: `Q_PLUGIN_METADATA(IID "tufao.Test/1.0")` and `Q_INTERFACES(Tufao::ClassHandler)`. The second macro must be exact, and is what Qt uses to know the class implements the [ClassHandler](#) interface. The first macro defines metadata about the plugin, specifically the IID of the plugin. The string can be anything meaningful, but is typically the now-common reverse resource locator used by so many languages. So if you were with the Stellarium project, the IID would be something like `org.stellarium.PluginName/1.0` where *PluginName* would be name of this specific plugin, and *1.0* would be the plugin version.

In the URL, the first path component is the object name (technically, if a context is being used, the context will be the first path component - see [ClassHandlerManager](#) - but that is transparent to implementing this class). In your constructor, you should call `setObjectName("forum")`; where the parameter is what you want matched in the URL path. For clarity, using the name of the class itself is a good idea. As it is case-sensitive, you may want to use the lower case version. Once this is done, incoming requests that match the object name will be dispatched (if they match a method; see below).

Implementing methods as end-points

The next path component is the name of a method in the class. The method name must match exactly, and is case-sensitive. The method should be a `public slot`, and the signature of the method is very important. The method must accept at least two parameters, and they must be `Tufao::HttpRequest` & `request`, `Tufao::HttpResponse` & `response`. The parameter names must be that exactly, as the dispatch uses parameter names of the methods to find the correct method to dispatch to. Further, the parameter types are used to convert the strings of the URL path to the data types the method requires.

As these two parameters must be on every method, the simplest method could have would have these two parameters. This allows you access to the request if you need it, and the response to write out the information back to the requester.

In the above example, the actual declaration of the method would be: `void read(Tufao::HttpRequest & request, Tufao::HttpResponse & response, QString forumName, QString threadName, int pageNumber)`. The parameter names are used to match the path components in the URL. This means that a URL will always have an *even number* of path components after the class name; half for the name of the parameters, and the other half for the values. If a value cannot be converted to the type of the parameter, the request is not handled, and a message logged. So the request would end up passing to the next registered `AbstractHttpRequestHandler` registered with the `HttpRequestRouter`.

If a context has been assigned to the `ClassHandlerManager` that is responsible for this plugin, it is also important to check to see if the request has a context. This is done simply with `!request.context().isEmpty()`. If the context is not empty, you may need to set it into the output, so that any URL's referenced (either in HTML or in Ajax) can prepend the context to their paths. If you are the sole user of your plugin, and you know what context you are going to use, you can avoid this and directly use the context in your HTML.

Finally, as the user has no control over instance lifetime (creation is done at application start for static plugins and load-time for dynamic ones, and objects are destroyed at application termination for static plugins and when the plugin is unloaded for dynamic ones), there are `void init()` and `void deinit()` methods that are called before an instance is dispatched to, and before shutdown. This allows for the creation or obtaining of resources that should not be attempted until the `QCoreApplication` is running, and that should be freed before the `QCoreApplication` is terminated.

Packaging

This class is also the superclass for plugins. One you subclass, you compile that code into either a dynamic or a static plugin. Dynamic plugins are a bit more robust in this case, and you can find an example project in the examples directory, named `sample_plugin`, that has a project with an executable as well as a dynamic plugin. You can use the `CMake` files for your own project.

If using a static plugins, you can load as many plugins as you like, but, they will all be connected to the same context (if used; see `ClassHandlerManager`). This is because there is not way to determine what the IID of the plugin is when loaded statically.

If using a dynamic plugin, the plugin must be installed into a path where it will be found. This includes all of the following:

- Every entry in the OS's library path with a `Tufao` sub-directory
- `/Library/Application Support/Tufao` on Mac.
- `~/Library/Application Support/Tufao` on Mac.
- `~/Tufao` on UNIX (non-Mac).
- The CWD of the application itself

Each of these paths is searched for a plugins subdirectory. Each entry in that directory that is a dynamic library is loaded if it implements the `ClassHandler` interface.

Since

1.2

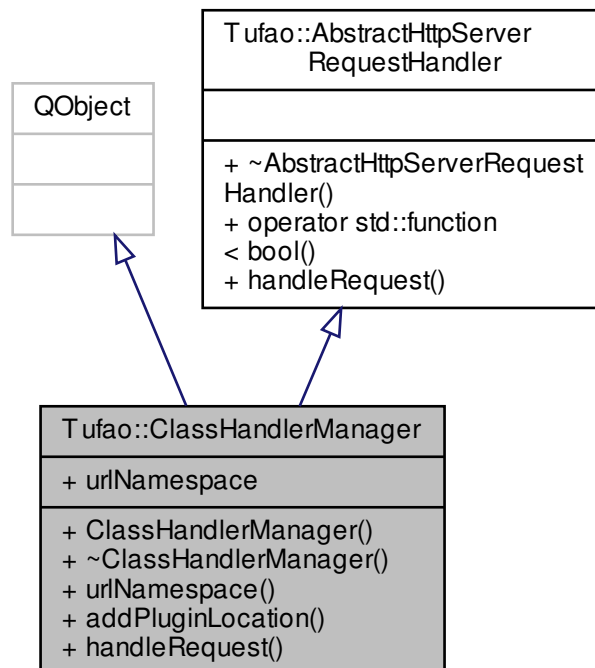
The documentation for this class was generated from the following file:

- classhandler.h

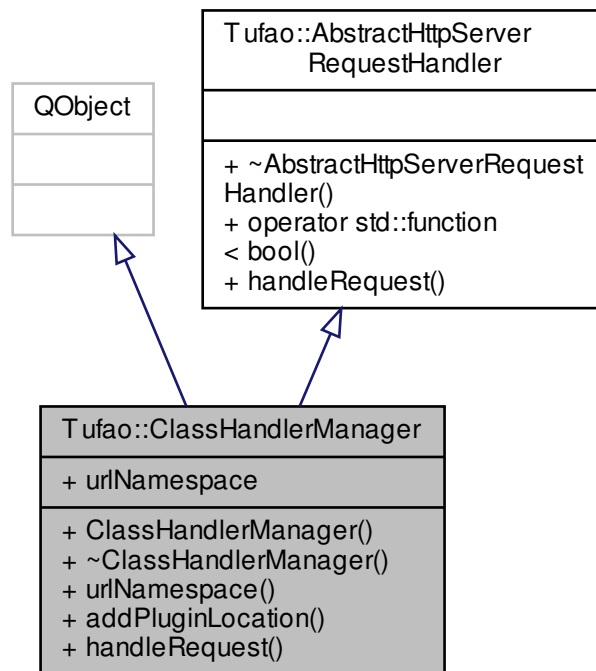
10.5 Tufao::ClassHandlerManager Class Reference

This class serves as the `HttpRequestHandler` for all [ClassHandler](#) plugins.

Inheritance diagram for `Tufao::ClassHandlerManager`:



Collaboration diagram for Tufao::ClassHandlerManager:



Public Slots

- bool **handleRequest** ([HttpServerRequest](#) &request, [HttpServerResponse](#) &response) override

Public Member Functions

- [ClassHandlerManager](#) (const QString &pluginID=QString{}, const QString &urlNamespace=QString{}, QObject *parent=0)
Constructs a [ClassHandlerManager](#) object.
- virtual [~ClassHandlerManager](#) ()
Destroys the object.
- QString **urlNamespace** () const

Static Public Member Functions

- static void [addPluginLocation](#) (const QString location)
Adds a non-standard path to the search paths.

Properties

- QString **urlNamespace**

10.5.1 Detailed Description

This class serves as the `HttpServerRequestHandler` for all [ClassHandler](#) plugins.

This class is used to manage one or more [ClassHandler](#) plugins. To use this class, you simply instantiate and register with the [HttpServerRequestRouter](#). You can specify that an instance of this class is only to manage plugins with a particular IID, and you can also assign a `urlNamespace` (the path's prefix of a URL).

If you specify a `urlNamespace` at creation time, only URL's whose path have that string as the prefix (case-sensitive) will receive dispatches. It's important to use a string that begins with "/" as the `urlNamespace`.

Also, requests whose url's path start with `urlNamespace`, but have some character different than '/' following are not considered to be within the same `urlNamespace` and will be ignored. This means that if you choose `"/forum"` as `urlNamespace`, requests to `"/forumb"` will be ignored. You're safe.

Static plugins will be registered by the first instance of this class created, regardless of if they specify an IID (as IID is not available at runtime for static plugins). Otherwise, only plugins that have a matching IID to the `pluginID` provided will be loaded by this instance.

To have a single instance load and register all plugins, simply leave the `pluginID` blank.

If `urlNamespace` is used, in your [ClassHandler](#) instances you will likely need to pass the value of the `urlNamespace` out into your response so that things like URL's in HTML or Ajax calls can prepend the `urlNamespace` to their paths; otherwise the calls will fail.

Since

1.2

10.5.2 Constructor & Destructor Documentation

10.5.2.1 `ClassHandlerManager()`

```
Tufao::ClassHandlerManager::ClassHandlerManager (
    const QString & pluginID = QString{},
    const QString & urlNamespace = QString{},
    QObject * parent = 0 ) [explicit]
```

Constructs a [ClassHandlerManager](#) object.

Parameters

<i>pluginID</i>	if provided, this instance will only load plugins whose IID matches.
<i>urlNamespace</i>	if provided, this is the <code>urlNamespace</code> (requests whose url's path start with <code>urlNamespace</code>) for the plugins managed by this instance. Before dispatching a request to one of the plugins, the context is checked, and only if it matches does the request get dispatched.
<i>parent</i>	is passed to the <code>QObject</code> constructor.

10.5.3 Member Function Documentation

10.5.3.1 addPluginLocation()

```
static void Tufao::ClassHandlerManager::addPluginLocation (
    const QString location ) [static]
```

Adds a non-standard path to the search paths.

By default, the standard locations are searched for plugins. The standard paths are the system library paths with a [Tufao](#) sub-directory, the executable path, and the applications configuration directory; on linux, `~/.tufao`, and on Mac both `/Library/Application Support/Tufao` and `~/Library/Application Support/Tufao`. Each of these directories is search to see if it has a plugins sud-directoy, and if it does, all dynamic libraries in each matching directroy is examined to determine if it is a plugin.

Parameters

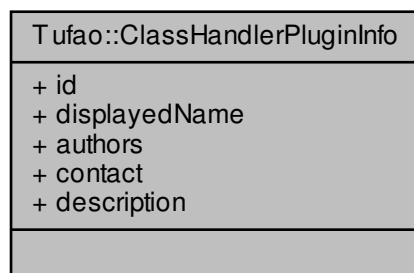
<i>location</i>	an absolute path to be added to the search paths for plugins.
-----------------	---

The documentation for this class was generated from the following file:

- classhandlermanager.h

10.6 Tufao::ClassHandlerPluginInfo Struct Reference

Collaboration diagram for Tufao::ClassHandlerPluginInfo:



Public Attributes

- [QString](#) [id](#)

The plugin ID. It MUST match the lib file name (case sensitive).

- QString [displayName](#)

The displayed name, e.g. "ServerSettings".

- QString [authors](#)

The comma separated list of authors, e.g. "Timothy Reaves".

- QString [contact](#)

The contact email or URL.

- QString [description](#)

The HTML description of the plugin.

10.6.1 Detailed Description

Contains information about a [ClassHandler](#) plugin.

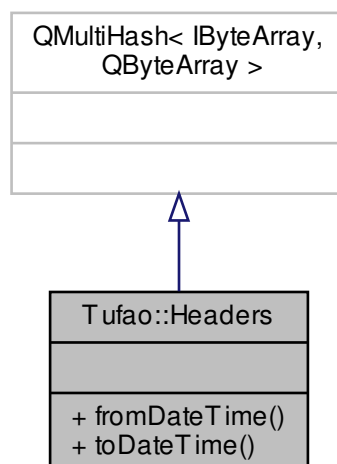
The documentation for this struct was generated from the following file:

- classhandler.h

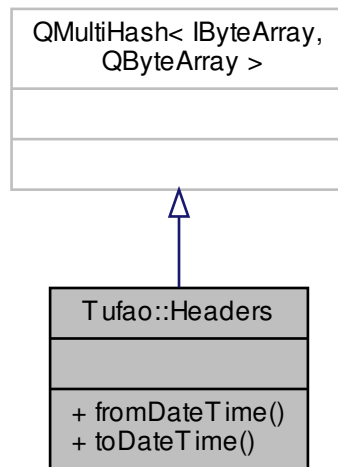
10.7 Tufao::Headers Struct Reference

This class provides a representation of HTTP headers.

Inheritance diagram for Tufao::Headers:



Collaboration diagram for Tufao::Headers:



Static Public Member Functions

- static TUFAO_EXPORT QByteArray [fromDateTime](#) (const QDateTime &dateTime)
Returns a RFC 1123 date time formatted string if `dateTime`.
- static TUFAO_EXPORT QDateTime [toDateTime](#) (const QByteArray &headerValue, const QDateTime &defaultValue=QDateTime())
Try to decode `headerValue` using the most common http date time formats.

10.7.1 Detailed Description

This class provides a representation of HTTP headers.

HTTP headers are string-based properties with case-insensitive keys.

See also

[Tufao::IByteArray](#)

10.7.2 Member Function Documentation

10.7.2.1 fromDateTime()

```
static TUFAO_EXPORT QByteArray Tufao::Headers::fromDateTime (
    const QDateTime & dateTime ) [static]
```

Returns a RFC 1123 date time formatted string if *dateTime*.

It's the standard date time format in HTTP.

Since

0.3

10.7.2.2 toDateTime()

```
static TUFAO_EXPORT QDateTime Tufao::Headers::toDateTime (
    const QByteArray & headerValue,
    const QDateTime & defaultValue = QDateTime() ) [static]
```

Try to decode *headerValue* using the most common http date time formats.

These formats are:

- RFC 1123
- RFC 1036
- ANSI C's `asctime()`

Returns

the converted QDateTime object or *defaultValue* if the conversion fails.

Since

0.3

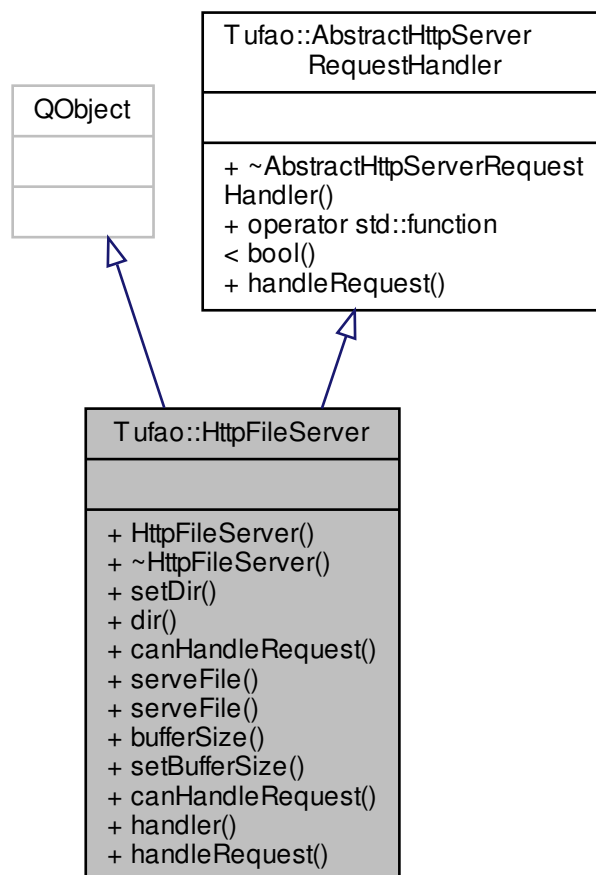
The documentation for this struct was generated from the following file:

- `headers.h`

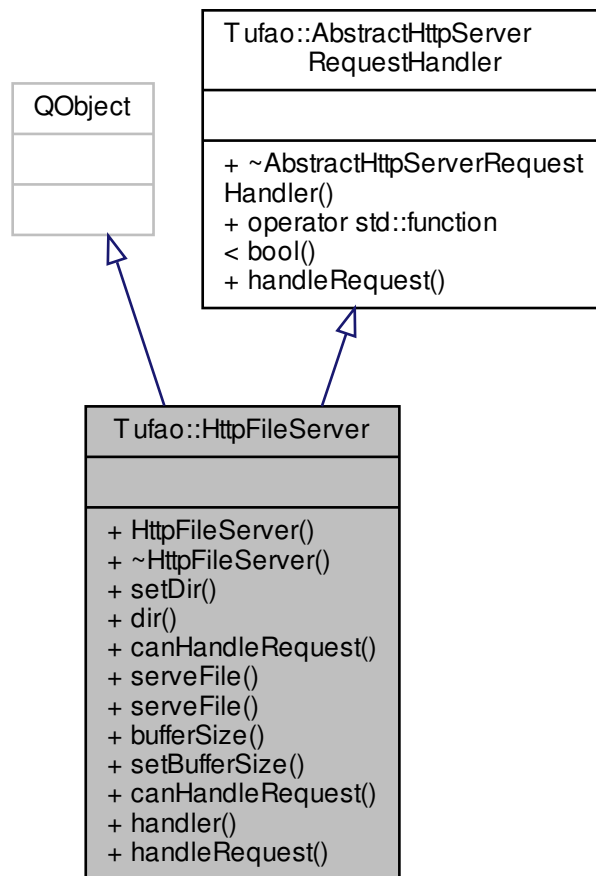
10.8 Tufao::HttpFileServer Class Reference

You can use this class to serve static files under Tufão.

Inheritance diagram for Tufao::HttpFileServer:



Collaboration diagram for Tufao::HttpFileServer:



Public Slots

- bool [handleRequest](#) ([Tufao::HttpServerRequest](#) &request, [Tufao::HttpServerResponse](#) &response) override
It searches for the file requested in the root dir and respond to the request, if the file is found.

Public Member Functions

- [HttpFileServer](#) (const QString &dir, QObject *parent=0)
Constructs a [HttpFileServer](#) object.
- [~HttpFileServer](#) ()
Destroys the object.
- void [setDir](#) (const QString &dir)
Set the root dir.
- QString [dir](#) () const
Return the root dir containing the files to be served.
- bool [canHandleRequest](#) (const [HttpServerRequest](#) &request)
Returns true iff [HttpFileServer::handleRequest](#) will return true.

Static Public Member Functions

- static void [serveFile](#) (const QString &fileName, [HttpServerRequest](#) &request, [HttpServerResponse](#) &response)
Analyze the request and serve the file pointed by filename.
- static bool [serveFile](#) (const QString &fileName, [HttpServerResponse](#) &response, [HttpResponseStatus](#) statusCode)
This member function doesn't serve any file, just set the response body to the contents in the file pointed by filename.
- static qint64 [bufferSize](#) ()
Return the buffer size used.
- static void [setBufferSize](#) (qint64 size)
Set the buffer size.
- static bool [canHandleRequest](#) (const [HttpServerRequest](#) &request, const QString &root)
Returns true iff [HttpFileServer::handleRequest](#) will return true.
- static std::function< bool([HttpServerRequest](#) &, [HttpServerResponse](#) &)> [handler](#) (const QString &rootDir)
Returns a handler that don't depends on another object.

10.8.1 Detailed Description

You can use this class to serve static files under Tufão.

It provides a robust HTTP file server, supporting conditional and byte-range requests.

The two main approaches are:

- Construct an object and use the [AbstractHttpServerRequestHandler](#) API implemented by [HttpFileServer](#)
- Use the static methods to serve a file (or set the entity in the response body)

The algorithm used to serve files will handle the following set of headers:

- If-Modified-Since
- If-Unmodified-Since
- If-Range
- Range
- Content-Type through QMimeDatabase (*Since version 1.0*)

It won't handle:

- ETag (If-Match and If-None-Match headers)
- Cache-Control response header: Useful for set cache max age
- Content-Disposition response header
- Content-MD5 response header

Since

0.3

10.8.2 Constructor & Destructor Documentation

10.8.2.1 `HttpFileServer()`

```
Tufao::HttpFileServer::HttpFileServer (
    const QString & dir,
    QObject * parent = 0 ) [explicit]
```

Constructs a [HttpFileServer](#) object.

`parent` is passed to the `QObject` constructor.

`dir` is used as root dir to serve files.

10.8.3 Member Function Documentation

10.8.3.1 `bufferSize()`

```
static qint64 Tufao::HttpFileServer::bufferSize ( ) [static]
```

Return the buffer size used.

When serving files, [HttpFileServer](#) allocates some bytes of the file in the memory before sending it to the network. The maximum space allocated is the buffer size. This method returns what number is this.

Note

The buffer size is global to all [HttpFileServer](#) objects.

See also

[setBufferSize](#)

10.8.3.2 `canHandleRequest()` [1/2]

```
bool Tufao::HttpFileServer::canHandleRequest (
    const HttpServerRequest & request )
```

Returns true iff [HttpFileServer::handleRequest](#) will return true.

Since

1.3

10.8.3.3 canHandleRequest() [2/2]

```
static bool Tufao::HttpFileServer::canHandleRequest (
    const HttpServerRequest & request,
    const QString & root ) [static]
```

Returns true iff [HttpFileServer::handleRequest](#) will return true.

Since

1.3

10.8.3.4 handler()

```
static std::function<bool(HttpServerRequest&, HttpServerResponse&)> Tufao::HttpFileServer↵
::handler (
    const QString & rootDir ) [static]
```

Returns a handler that don't depends on another object.

The purpose of this alternative handler is to free you of the worry of maintain the [HttpFileServer](#)'s object (lifetime) while the functor object is being used.

Parameters

<i>rootDir</i>	The root dir to serve files.
----------------	------------------------------

Since

1.0

10.8.3.5 handleRequest

```
bool Tufao::HttpFileServer::handleRequest (
    Tufao::HttpServerRequest & request,
    Tufao::HttpServerResponse & response ) [override], [slot]
```

It searches for the file requested in the root dir and respond to the request, if the file is found.

Note

This method won't let requests access files outside the root dir folder and should be preferred over self-made implementations, as its safer.

Since

1.0

10.8.3.6 `serveFile()` [1/2]

```
static void Tufao::HttpFileServer::serveFile (
    const QString & fileName,
    HttpRequest & request,
    HttpResponse & response ) [static]
```

Analyze the request and serve the file pointed by filename.

Since

1.0

10.8.3.7 `serveFile()` [2/2]

```
static bool Tufao::HttpFileServer::serveFile (
    const QString & fileName,
    HttpResponse & response,
    HttpResponseStatus statusCode ) [static]
```

This member function doesn't serve any file, just set the response body to the contents in the file pointed by filename.

It's useful in some scenarios, like serving 404-pages.

Since

1.0

10.8.3.8 `setDir()`

```
void Tufao::HttpFileServer::setDir (
    const QString & dir )
```

Set the root dir.

The root dir is the dir containing the files to be served by the [HttpFileServer](#) object.

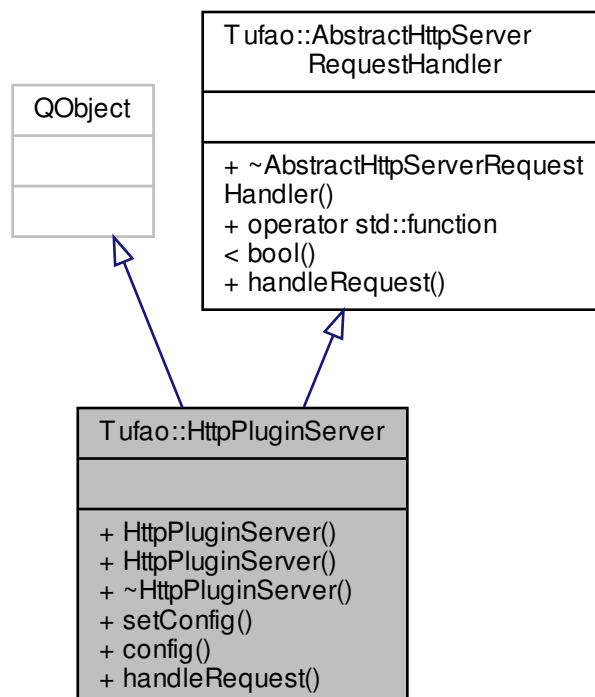
The documentation for this class was generated from the following file:

- `httpfileserver.h`

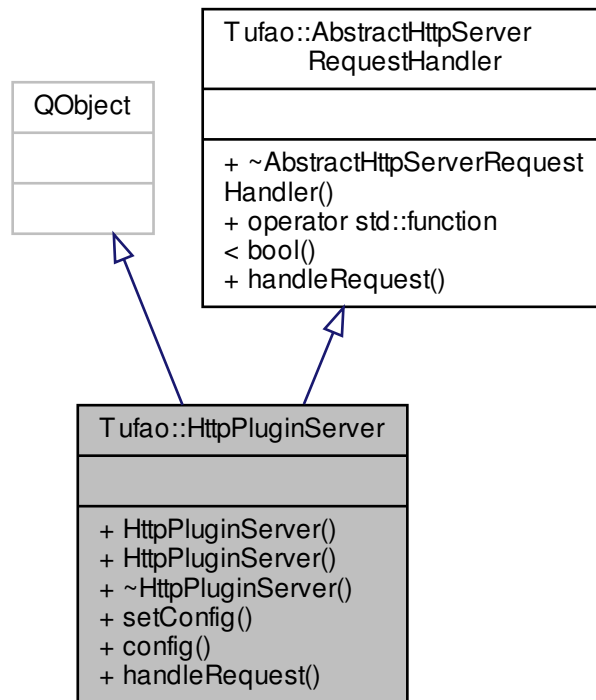
10.9 Tufao::HttpPluginServer Class Reference

This class provides a plugin-based request handler.

Inheritance diagram for Tufao::HttpPluginServer:



Collaboration diagram for Tufao::HttpPluginServer:



Public Slots

- bool `handleRequest` (`Tufao::HttpServerRequest` &request, `Tufao::HttpServerResponse` &response) override
Handle the request using the loaded plugins and rules.

Public Member Functions

- `HttpPluginServer` (`QObject` *parent=0)
Constructs a null `HttpPluginServer` object.
- `HttpPluginServer` (const `QString` &configFile, `QObject` *parent=0)
Constructs a `HttpPluginServer` object.
- `~HttpPluginServer` ()
Destroys the object.
- bool `setConfig` (const `QString` &file)
Set the configuration file used to handle requests.
- `QString` `config` () const
Returns the path of the last configuration file used.

10.9.1 Detailed Description

This class provides a plugin-based request handler.

Use it if you need to change the application code without restart the server.

It maintains its own set of rules and is as powerful as [HttpServerRequestRouter](#) (and internally uses it), but, in contrast, exports its configuration through a json file and will use handlers loaded from plugins.

The file format is described in the [HttpPluginServer::setConfig](#) method.

Note

The object will monitor the config file for changes and reload it as changes happens.

See also

[AbstractHttpServerRequestHandlerFactory](#) to implement your plugins.
[Tufão's plugin system](#)

Since

0.3

10.9.2 Constructor & Destructor Documentation

10.9.2.1 HttpPluginServer() [1/2]

```
Tufao::HttpPluginServer::HttpPluginServer (  
    QObject * parent = 0 ) [explicit]
```

Constructs a null [HttpPluginServer](#) object.

`parent` is passed to the `QObject` constructor.

10.9.2.2 HttpPluginServer() [2/2]

```
Tufao::HttpPluginServer::HttpPluginServer (  
    const QString & configFile,  
    QObject * parent = 0 ) [explicit]
```

Constructs a [HttpPluginServer](#) object.

`parent` is passed to the `QObject` constructor.

`configFile` is used as configuration file.

See also

[setConfig](#)

10.9.3 Member Function Documentation

10.9.3.1 `config()`

```
QString Tufao::HttpPluginServer::config ( ) const
```

Returns the path of the last configuration file used.

This file is used to handle requests, loading the appropriate plugins, generating actual handlers and mapping them to the rules described in this file.

See also

[setConfig](#)

10.9.3.2 `handleRequest`

```
bool Tufao::HttpPluginServer::handleRequest (
    Tufao::HttpRequest & request,
    Tufao::HttpResponse & response ) [override], [slot]
```

Handle the request using the loaded plugins and rules.

Note

In Tufão 0.x, `handleRequest` caught exceptions thrown by plugins, but in Tufão 1.0, this behaviour is not used anymore, because it was stealing the power and the control of the programmer.

Since

1.0

10.9.3.3 `setConfig()`

```
bool Tufao::HttpPluginServer::setConfig (
    const QString & file )
```

Set the configuration file used to handle requests.

After the file is set, [HttpPluginServer](#) will watch the file for changes and reload its config when the file changes.

Warning

[HttpPluginServer](#) can't monitor file links correctly.

Note

The old config is cleared even if it fails to set the new config.

Return values

<i>true</i>	if HttpPluginServer finds the file.
<i>false</i>	if HttpPluginServer can't find the file.

Call this method with an empty string if you want to *clear* the plugin server.

The [HttpPluginServer](#) behaviour

An simplified use case to describing how [HttpPluginServer](#) reacts to changes follows:

1. You start with a default-constructed [HttpPluginServer](#)
2. You use `setConfig` with an inexistent file
 - (a) The [HttpPluginServer](#) do not find the file
 - (b) [HttpPluginServer::setConfig](#) returns false
 - (c) [HttpPluginServer](#) object remains in the previous state
3. You use `setConfig` with a invalid file
 - (a) [HttpPluginServer](#) starts to monitor the config file
 - (b) [HttpPluginServer::setConfig](#) returns true
 - (c) [HttpPluginServer](#) reads the invalid file and remains in the previous state.
4. You fill the config file with a valid config.
 - (a) [HttpPluginServer](#) object load the new contents
 - (b) [HttpPluginServer](#) try to load every plugin and fill the router. If a plugin cannot be loaded, it will be skipped and a warning message is sent through `qWarning`. If you need to load this plugin, make any modification to the config file and [HttpPluginServer](#) will try again.
5. You fill the config file with an invalid config.
 - (a) [HttpPluginServer](#) see and ignores the changes, remaining with the previous settings.
6. You remove the config file.
 - (a) [HttpPluginServer](#) object come back to the default-constructed state.

version: 0

If the last config had "version: 0", then it means no more monitoring either (this is what default-constructed state means).

version: 1

If the last config had "version: 1", then [HttpPluginServer](#) will (after the cleanup) start to monitor the containing folder, waiting until a config file with the same name is available again to resume its operation.

Note

A later call to [HttpPluginServer::setConfig](#) can be used to stop the monitoring.

Note

If the containing dir is also erased, [HttpPluginServer](#) can do nothing and the monitoring will stop.

The file format

The configuration file format is json-based. If you aren't used to JSON, read the [json specification](#).

Note

The old Tufão 0.x releases used a file with the syntax based on the QSettings ini format and forced you to use the *tufao-routes-editor* application to edit this file.

The file must have a root json object with 3 attributes:

- *version*: It must indicate the version of the configuration file. The list of acceptable values are:
 - *0*: Version recognizable by Tufão 1.x, starting from 1.0
 - *1*: Version recognizable by Tufão 1.x, starting from 1.2. The only difference is the autoreloading behaviour. If you delete the config file, Tufão will start to monitor the containing folder and resume the normal operation as soon as the file is added to the folder again.
- *plugins*: This attribute stores metadata about the plugins. All plugins specified here will be loaded, even if they aren't used in the request router. The value of this field must be an array and each element of this array must be an object with the following attributes:
 - *name*: This is the name of the plugin and defines how you will refer to this plugin later. You can't have two plugins with the same name. This attribute is **required**.
 - *path*: This is the path of the plugin in the filesystem. Relative paths are supported, and are relative to the configuration file. This attribute is **required**.
 - *dependencies*: This field specifies a list of plugins that must be loaded before this plugin. This plugin will be capable of access plugins listed here. This attribute is **optional**.
 - *customData*: It's a field whose value is converted to a QVariant and passed to the plugin. It can be used to pass arbitrary data, like application name or whatever. This attribute is **optional**.
- *requests*: This attribute stores metadata about the requests handled by this object. The value of this field is an array and each element of this array describes a handler and is an object with the following attributes:
 - *path*: Defines the regex pattern used to filter requests based on the url's path component. The regex is processed through QRegularExpression. This attribute is **required** and **must** be an valid regex.
 - *plugin*: Defines what plugin is used to handle request matching the rules defined in this containing block. This attribute is **required**.
 - *method*: Define what HTTP method is accepted by this handler. This field is **optional** and, if it's not defined, it won't be used to filter the requests.

Note

An empty value isn't acceptable to a field, except for *customValue* at *plugins* and *path* at *requests* fields. If you use an unacceptable value, Tufão may reject your file.

An example follows:

```
{
  version: 1,
  plugins: [
    {
      name: "home",
      path: "/home/vinipsmaker/Projetos/tufao-project42/build/plugins/libhome.so",
      customData: {appName: "Hello World", root: "/"},
    },
  ],
}
```

```

    {
        name: "user",
        path: "show_user.so",
        dependencies: ["home"]
    },
    {
        name: "404",
        path: "/usr/lib/tufao/plugins/notfound.so",
        customData: "<h1>Not Found</h1><p>I'm sorry, but it's your fault</p>"
    }
],
requests: [
    {
        path: "^/$",
        plugin: "home",
        method: "GET"
    },
    {
        path: "^/user/(\w*)$",
        plugin: "user"
    },
    {
        path: "",
        plugin: "404"
    }
]
}

```

The *requests* attribute is used to seed data to a [HttpServerRequestRouter](#) object. Because this, you can use features like return false from a handler to allow another handler handle a request.

Note

If the [HttpPluginServer](#) finds an attribute not recognizable, the attribute will be skipped. You can use this to extend the file with customized fields and the [HttpPluginServer](#) will continue to behave normally.

Warning

After reload the config, Tufão might recycle some objects to achieve the goal of build a matching request router faster, if dependencies don't change. You can rely on the dependency system for initialization order, but you can't know if the plugins will or not be reloaded after the config changes, because the behaviour is not defined and might change in different Tufão versions.

See also

[config](#)

The documentation for this class was generated from the following file:

- `httppluginserver.h`

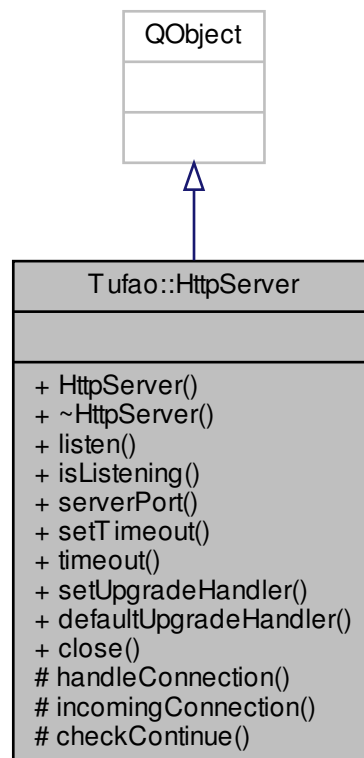
10.10 Tufao::HttpServer Class Reference

The [Tufao::HttpServer](#) class provides an implementation of the HTTP protocol.

Inheritance diagram for Tufao::HttpServer:



Collaboration diagram for Tufao::HttpServer:



Public Types

- typedef std::function< void([HttpServerRequest](#) &request, const QByteArray &)> [UpgradeHandler](#)

A typedef to http upgrade request handler.

Public Slots

- void [close](#) ()

Closes the server.

Signals

- void [requestReady](#) ([Tufao::HttpServerRequest](#) &request, [Tufao::HttpServerResponse](#) &response)

This signal is emitted each time there is request.

Public Member Functions

- [HttpServer](#) (QObject *parent=0)
Constructs a [Tufao::HttpServer](#) object.
- [~HttpServer](#) ()
Destroys the object.
- bool [listen](#) (const QHostAddress &address=QHostAddress::Any, quint16 port=0)
*Tells the server to listen for incoming connections on address *address* and port *port*.*
- bool [isListening](#) () const
Returns true if the server is listening for incoming connections.
- quint16 [serverPort](#) () const
Returns the server's port if the server is listening; otherwise returns 0.
- void [setTimeout](#) (int msec=0)
*Sets the timeout of new connections to *msec* milliseconds.*
- int [timeout](#) () const
Returns the current set timeout.
- void [setUpgradeHandler](#) (UpgradeHandler functor)
This method sets the handler that will be called to handle http upgrade requests.

Static Public Member Functions

- static [UpgradeHandler](#) [defaultUpgradeHandler](#) ()
Returns the default http upgrade request's handler.

Protected Member Functions

- void [handleConnection](#) (QAbstractSocket *connection)
*Call this function will make [Tufao::HttpServer](#) handle the connection *connection*.*
- virtual void [incomingConnection](#) (qintptr socketDescriptor)
This virtual function is called by [HttpServer](#) when a new connection is available.
- virtual void [checkContinue](#) (HttpServerRequest &request, HttpServerResponse &response)
This virtual function is called by [HttpServer](#) when a client do a request with the HTTP header "Expect: 100-continue".

10.10.1 Detailed Description

The [Tufao::HttpServer](#) class provides an implementation of the HTTP protocol.

The HTTP is a stateless request-response based protocol. It let you create distributed dynamic collaborative applications.

To create a webserver, all you need to do is call [Tufao::HttpServer::listen](#) and handle the [Tufao::HttpServer::requestReady](#) signal.


```
#include <Tufao/HttpServer>

class WebServer: public QObject
{
    Q_OBJECT
public:
    explicit WebServer(QObject *parent = NULL) :
        QObject(parent),
        httpServer(new Tufao::HttpServer(this))
    {
        using namespace Tufao;
        connect(httpServer, &HttpServer::requestReady,
            [](HttpServerRequest &request,
            HttpServerResponse &response) {
                response.writeHead(200, "OK");
                response.headers().insert("Content-Type", "text/plain");
                response.write("Hello World\n");
                response.end();
            });

        httpServer->listen(QHostAddress::Any, 8080);
    }

private:
    Tufao::HttpServer *httpServer;
};
```

See also

[Tufao::HttpServerRequest](#)
[Tufao::HttpServerResponse](#)

10.10.2 Member Typedef Documentation**10.10.2.1 UpgradeHandler**

```
typedef std::function<void(HttpServerRequest &request, const QByteArray&)> Tufao::HttpServer::UpgradeHandler
```

A typedef to http upgrade request handler.

See also

[AbstractHttpUpgradeHandler](#) [setUpgradeHandler](#)

Since

1.0

10.10.3 Constructor & Destructor Documentation**10.10.3.1 HttpServer()**

```
Tufao::HttpServer::HttpServer (
    QObject * parent = 0 ) [explicit]
```

Constructs a [Tufao::HttpServer](#) object.

`parent` is passed to the `QObject` constructor.

10.10.4 Member Function Documentation

10.10.4.1 `checkContinue()`

```
virtual void Tufao::HttpServer::checkContinue (
    HttpServerRequest & request,
    HttpServerResponse & response ) [protected], [virtual]
```

This virtual function is called by [HttpServer](#) when a client do a request with the HTTP header "Expect: 100-continue".

The base implementation call `Tufao::HttpServerRequest::writeContinue` and emit the [Tufao::HttpServer::requestReady](#) signal.

Reimplement this function to alter the server's behavior when a "Expect: 100-continue" request is received.

Note

Don't delete the request or the response object, they will be deleted when the connection closes. If you need delete them before, just close the connection or call the `QObject::deleteLater`.

Since

1.0

10.10.4.2 `close`

```
void Tufao::HttpServer::close ( ) [slot]
```

Closes the server.

The server will no longer listen for incoming connections.

10.10.4.3 `defaultUpgradeHandler()`

```
static UpgradeHandler Tufao::HttpServer::defaultUpgradeHandler ( ) [static]
```

Returns the default http upgrade request's handler.

The default handler closes the connection.

Since

1.0

10.10.4.4 `handleConnection()`

```
void Tufao::HttpServer::handleConnection (
    QAbstractSocket * connection ) [protected]
```

Call this function will make [Tufao::HttpServer](#) handle the connection `connection`.

The [Tufao::HttpServer](#) object will take ownership of the `connection` object and delete it when appropriate.

10.10.4.5 `incomingConnection()`

```
virtual void Tufao::HttpServer::incomingConnection (
    qintptr socketDescriptor ) [protected], [virtual]
```

This virtual function is called by [HttpServer](#) when a new connection is available.

The base implementation creates a QTcpSocket, sets the socket descriptor and call [Tufao::HttpServer::handleConnection](#).

Reimplement this function to alter the server's behavior when a connection is available.

10.10.4.6 `listen()`

```
bool Tufao::HttpServer::listen (
    const QHostAddress & address = QHostAddress::Any,
    quint16 port = 0 )
```

Tells the server to listen for incoming connections on address `address` and port `port`.

If `port` is 0, a port is chosen automatically. The default registered port to HTTP server is 80.

If `address` is `QHostAddress::Any`, the server will listen on all network interfaces.

Returns

`true` on success

See also

[Tufao::HttpServer::isListening](#) [Tufao::HttpServer::serverPort](#)

10.10.4.7 `requestReady`

```
void Tufao::HttpServer::requestReady (
    Tufao::HttpServerRequest & request,
    Tufao::HttpServerResponse & response ) [signal]
```

This signal is emitted each time there is request.

Note

There may be multiple requests per connection (in the case of keep-alive connections) and [HttpServer](#) re-utilizes `request` objects, so you can't, as an example, create a map using `request` as key to identify sessions.

Warning

You MUST NOT delete `request` and `response`. `request` and `response` are deleted when the connection closes. Additionally, `response` will also be deleted when you are done with it (eg., calling [Tufao::HttpServerResponse::end](#)).

Note

If this is a POST request for a big file, you should increase the timeout for this individual request.

Parameters

<i>request</i>	An instance of Tufao::HttpServerRequest
<i>response</i>	An instance of Tufao::HttpServerResponse

Since

1.0

10.10.4.8 `serverPort()`

```
quint16 Tufao::HttpServer::serverPort ( ) const
```

Returns the server's port if the server is listening; otherwise returns 0.

See also

[Tufao::HttpServer::listen](#) [Tufao::HttpServer::isListening](#)

10.10.4.9 `setTimeout()`

```
void Tufao::HttpServer::setTimeout (
    int msec = 0 )
```

Sets the timeout of new connections to `msec` milliseconds.

If you set the timeout to 0, then timeout feature will be disabled. You should NOT disable this feature to help to protect against DoS attacks.

The default timeout is 2 minutes (120000 milliseconds).

Note

You should call this function before [Tufao::HttpServer::listen](#).

10.10.4.10 setUpgradeHandler()

```
void Tufao::HttpServer::setUpgradeHandler (
    UpgradeHandler functor )
```

This method sets the handler that will be called to handle http upgrade requests.

Note

The connection object associated with request parameter ([Tufao::HttpRequest::socket](#)) will be deleted when disconnected. If you need to delete it sooner, just call `QIODevice::close` or `QObject::deleteLater`.

If you pass an empty `std::function` object, this function does nothing.

See also

[defaultUpgradeHandler](#)

Since

1.0

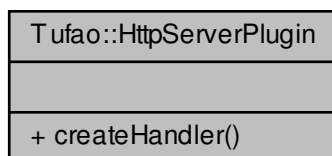
The documentation for this class was generated from the following file:

- `httpserver.h`

10.11 Tufao::HttpServerPlugin Class Reference

This class provides a factory interface to create request handlers and communicate with factories of other plugins.

Collaboration diagram for Tufao::HttpServerPlugin:

**Public Member Functions**

- `virtual std::function< bool(HttpRequest &, HttpResponse &)> createHandler` (const `QHash<QString, HttpServerPlugin *>` &dependencies, const `QVariant` &customData=`QVariant()`)=0
Creates a persistent handler.

10.11.1 Detailed Description

This class provides a factory interface to create request handlers and communicate with factories of other plugins.

An example follows:

```
#ifndef PLUGIN_H
#define PLUGIN_H

#include <Tufao/HttpServerPlugin>

class Plugin: public QObject, Tufao::HttpServerPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID TUFAO_HTTPSERVERPLUGIN_IID)
    Q_INTERFACES(Tufao::HttpServerPlugin)
public:
    std::function<bool(Tufao::HttpServerRequest&, Tufao::HttpServerResponse&)>
    createHandler(const QHash<QString, Tufao::HttpServerPlugin*> &dependencies,
                  const QVariant &customData = QVariant()) override;
};

#endif // PLUGIN_H
```

And its implementation file:

```
#include "plugin.h"
#include <QtCore/QtPlugin>
#include <Tufao/HttpServerResponse>

using namespace Tufao;

std::function<bool(HttpServerRequest&, HttpServerResponse&)>
Plugin::createHandler(const QHash<QString, HttpServerPlugin*> &,
                      const QVariant &)
{
    return [](HttpServerRequest &, HttpServerResponse &res){
        res.writeHead(HttpStatus::OK);
        res << "Responding from a evil plugin\n";
        res.end();
        return true;
    };
}
```

See also

[TUFAO_HTTPSERVERPLUGIN_IID](#)

Since

1.0

10.11.2 Member Function Documentation

10.11.2.1 createHandler()

```
virtual std::function<bool(HttpServerRequest&, HttpServerResponse&)> Tufao::HttpServerPlugin↵
::createHandler (
    const QHash< QString, HttpServerPlugin *> & dependencies,
    const QVariant & customData = QVariant() ) [pure virtual]
```

Creates a persistent handler.

Note

The handler created will be used as argument in [HttpServerRequestRouter::map](#). If you want to use a different handler to every request, you should create another handler in the body of the returned handler.

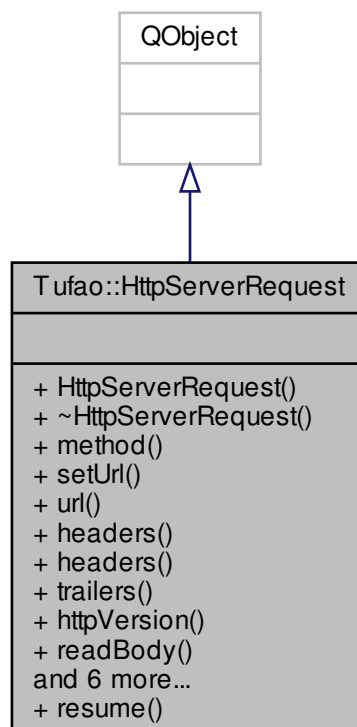
The documentation for this class was generated from the following file:

- [httpserverplugin.h](#)

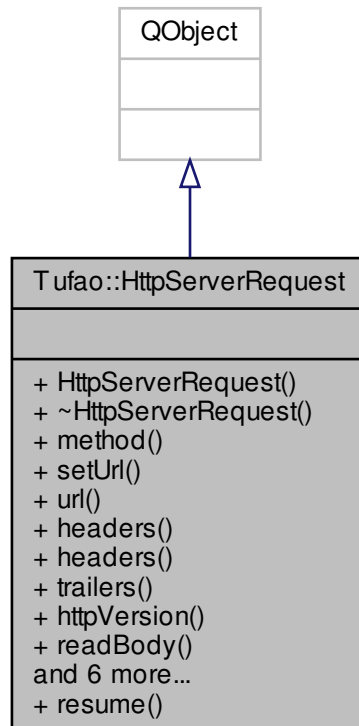
10.12 Tufao::HttpRequest Class Reference

The [Tufao::HttpServer](#) represents a HTTP request received by [Tufao::HttpServer](#).

Inheritance diagram for Tufao::HttpRequest:



Collaboration diagram for Tufao::HttpRequest:



Public Slots

- void `resume` ()

This function exists to support HTTP pipelining.

Signals

- void `ready` ()

This signal is emitted when most of the data about the request is available.

- void `data` ()

This signal is emitted each time a piece of the message body is received.

- void `end` ()

This signal is emitted exactly once for each request.

- void `close` ()

This signal is emitted when the underlying connection is closed (also caused by invalid requests).

- void `upgrade` ()

This signal is emitted when a http upgrade is requested.

Public Member Functions

- [HttpRequest](#) (QAbstractSocket &[socket](#), QObject *parent=0)
Constructs a [Tufao::HttpRequest](#) object.
- [~HttpRequest](#) ()
Destroys the object.
- QByteArray [method](#) () const
The request method.
- void [setUrl](#) (const QUrl &[url](#))
Sets the request URL.
- QUrl [url](#) () const
The request URL.
- Headers [headers](#) () const
The HTTP headers sent by the client.
- Headers & [headers](#) ()
The HTTP headers sent by the client.
- Headers [trailers](#) () const
The HTTP trailers (if present).
- HttpVersion [httpVersion](#) () const
Returns the HTTP protocol version used in the request.
- QByteArray [readBody](#) ()
Read the request's body.
- QAbstractSocket & [socket](#) () const
The QAbstractSocket object associated with the connection.
- void [setTimeout](#) (int msec=0)
*Sets the timeout of new connections to *msec* milliseconds.*
- int [timeout](#) () const
Returns the current set timeout.
- Tufao::HttpRequest::Options [responseOptions](#) () const
Returns the options obje that should be passed to the [Tufao::HttpRequest](#) constructor.
- QVariant [customData](#) () const
*Returns the user data as set in *setCustomData*.*
- void [setCustomData](#) (const QVariant &[data](#))
*Sets the custom data to *data*.*

Friends

- struct [Tufao::HttpRequest::Priv](#)

10.12.1 Detailed Description

The [Tufao::HttpRequest](#) represents a HTTP request received by [Tufao::HttpRequest](#).

Note

You can use it to create your own HTTP servers too, just handle the connections steps and pass the connection objects to [Tufao::HttpRequest](#) constructor. This may be useful if you want to create a threaded web server or use non conventional streams instead tcp sockets.

See also

[Tufao::HttpRequest](#)

10.12.2 Constructor & Destructor Documentation

10.12.2.1 `HttpServerRequest()`

```
Tufao::HttpServerRequest::HttpServerRequest (
    QAbstractSocket & socket,
    QObject * parent = 0 ) [explicit]
```

Constructs a [Tufao::HttpServerRequest](#) object.

`parent` is passed to the `QObject` constructor.

Parameters

<code>socket</code>	The connection used by Tufao::HttpServerRequest to receive HTTP messages. If you pass NULL, the object will be useless.
---------------------	---

Since

1.0

10.12.3 Member Function Documentation

10.12.3.1 `close`

```
void Tufao::HttpServerRequest::close ( ) [signal]
```

This signal is emitted when the underlying connection is closed (also caused by invalid requests).

Just like [Tufao::HttpServerRequest::end](#), this signal is emitted only once per request, and no more data signals will fire afterwards.

10.12.3.2 `customData()`

```
QVariant Tufao::HttpServerRequest::customData ( ) const
```

Returns the user data as set in `setCustomData`.

Note

This data will be erased upon a new request.

See also

[setCustomData](#)

Since

1.0

10.12.3.3 data

```
void Tufao::HttpRequest::data ( ) [signal]
```

This signal is emitted each time a piece of the message body is received.

Use [readBody\(\)](#) to consume the data.

Note

This signal is unsafe (read this: [Safe signals](#))!

Since

1.0

10.12.3.4 end

```
void Tufao::HttpRequest::end ( ) [signal]
```

This signal is emitted exactly once for each request.

Use [readBody\(\)](#) to access the request's body and [trailers\(\)](#) to access the headers sent after the body.

After that, no more data signals will be emitted for this session. A new session (if any) will be only initiated after you respond the request.

10.12.3.5 headers() [1/2]

```
Headers Tufao::HttpRequest::headers ( ) const
```

The HTTP headers sent by the client.

These headers are fully populated when the signal [Tufao::HttpRequest::ready](#) signal is emitted.

See also

[Tufao::HttpRequest::trailers\(\)](#)

10.12.3.6 headers() [2/2]

`Headers& Tufao::HttpServerRequest::headers ()`

The HTTP headers sent by the client.

These headers are fully populated when the signal `Tufao::HttpServerRequest::ready` signal is emitted.

See also

[Tufao::HttpServerRequest::trailers\(\)](#)

Since

0.3

10.12.3.7 method()

`QByteArray Tufao::HttpServerRequest::method () const`

The request method.

It can assume the following values:

- "HEAD"
- "GET"
- "POST"
- "PUT"
- "DELETE"
- "TRACE"
- "OPTIONS"
- "CONNECT"
- "PATCH"
- "COPY"
- "LOCK"
- "MKCOL"
- "MOVE"
- "PROPFIND"
- "PROPPATCH"
- "SEARCH"
- "UNLOCK"

- "REPORT"
- "MKACTIVITY"
- "CHECKOUT"
- "MERGE"
- "M-SEARCH"
- "NOTIFY"
- "SUBSCRIBE"
- "UNSUBSCRIBE"
- "PURGE"

10.12.3.8 readBody()

```
QByteArray Tufao::HttpRequest::readBody ( )
```

Read the request's body.

Returns

This request's object will buffer every piece of body received. After call this function, the buffered content is returned and the buffer is cleared.

Note

If you only call this function after [end\(\)](#) signal, the returned object will be the entire body of the request. Call this function when the [data\(\)](#) signal is emitted and save the body to disk if you expect to receive requests with bodies larger than available RAM.

See also

[data\(\)](#) [end\(\)](#)

Since

1.0

10.12.3.9 ready

```
void Tufao::HttpServerRequest::ready ( ) [signal]
```

This signal is emitted when most of the data about the request is available.

After this signal is emitted, you can safely interpret the request and the only missing parts may be (if any) the message body and the trailers.

Note

This signal is unsafe (read this: [Safe signals](#))!

Warning

Right before emit this signal, [HttpServerRequest](#) object will disconnect any slot connected to the signals listed below. This behaviour was chosen to allow you to think about the single HTTP session without worry about the Tufão behaviour of reuse the same objects to the same connections.

- [HttpServerRequest::data](#)
- [HttpServerRequest::end](#)

See also

[Tufao::HttpServerRequest::responseOptions](#) [Tufao::HttpServerRequest::data](#) [Tufao::HttpServerRequest::end](#)

Since

0.2

10.12.3.10 responseOptions()

```
Tufao::HttpServerResponse::Options Tufao::HttpServerRequest::responseOptions ( ) const
```

Returns the options obje that should be passed to the [Tufao::HttpServerResponse](#) constructor.

Since

0.2

10.12.3.11 resume

```
void Tufao::HttpRequest::resume ( ) [slot]
```

This function exists to support HTTP pipelining.

HTTP protocol allows several requests to be sent before the reply to the first one is issued. [HttpRequest](#) will stop processing its internal buffer once the end of some message is reached. You should call this function once you issue the reply to a message.

[HttpServer](#) will automatically connect the [HttpRequest::finished](#) signal to this slot, so you shouldn't need to worry about anything if you're using Tufão's standard abstractions.

Since

1.4

10.12.3.12 setCustomData()

```
void Tufao::HttpRequest::setCustomData (
    const QVariant & data )
```

Sets the custom data to `data`.

The custom data is a convenience method to allow users of [HttpRequest](#) to store some data in some requests. It's used in [Tufao::HttpRequestRouter](#) to pass the list of captured texts in the url to the subsequent handlers.

Note

This data will be erased upon a new request.

Since

1.0

10.12.3.13 setTimeout()

```
void Tufao::HttpRequest::setTimeout (
    int msec = 0 )
```

Sets the timeout of new connections to `msec` milliseconds.

The connection will be closed when no bytes are received during `msec` milliseconds.

If you set the timeout to 0, then timeout feature will be disabled.

By default, there is no timeout.

You can call this function at any time.

10.12.3.14 `setUrl()`

```
void Tufao::HttpServerRequest::setUrl (
    const QUrl & url )
```

Sets the request URL.

See also

[UrlRewriterHandler](#)

Since

1.0

10.12.3.15 `socket()`

```
QAbstractSocket& Tufao::HttpServerRequest::socket ( ) const
```

The QAbstractSocket object associated with the connection.

This will be a QTcpSocket object if created by [Tufao::HttpServer](#) and a QSslSocket if created by [Tufao::HttpsServer](#).

Since

1.0

10.12.3.16 `trailers()`

```
Headers Tufao::HttpServerRequest::trailers ( ) const
```

The HTTP trailers (if present).

Only populated after the [Tufao::HttpServerRequest::end](#) signal.

Trailers are headers sent after the body. Some headers can't be computed before the full body is generated. The solution to decrease the network latency is send the body before the associated metadata using the trailers technique.

10.12.3.17 upgrade

```
void Tufao::HttpServerRequest::upgrade ( ) [signal]
```

This signal is emitted when a http upgrade is requested.

Note

If this signal is emitted, then the signals [HttpServerRequest::ready](#), [HttpServerRequest::end](#) and [HttpServerRequest::close](#) won't be emitted.

The body is set to the initial bytes from the new connection session (under the new protocol).

Since

1.0

10.12.3.18 url()

```
QUrl Tufao::HttpServerRequest::url ( ) const
```

The request URL.

This contains only the URL that is present in the actual HTTP request. If the request is:

```
GET /login?username=tux HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

Then [Tufao::HttpServerRequest::url\(\)](#) will be constructed with `"/login?username=tux"`.

Since

1.0

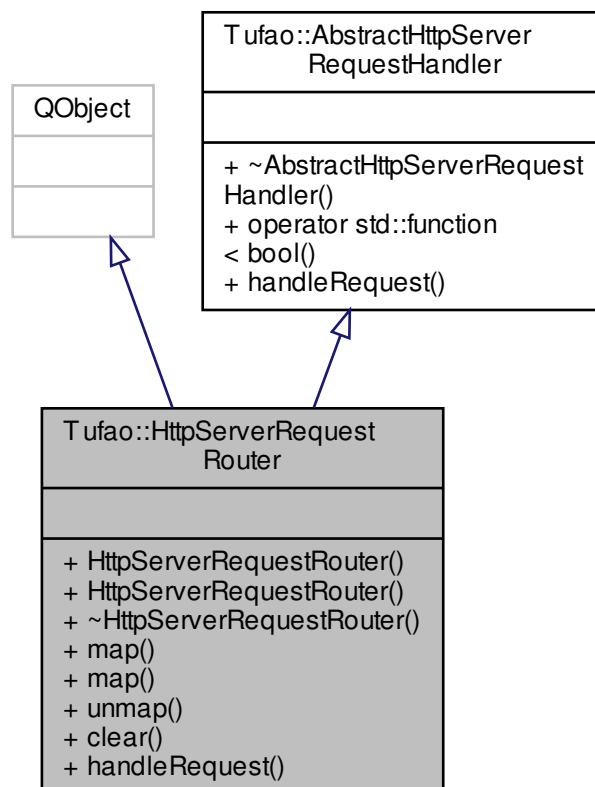
The documentation for this class was generated from the following file:

- `httpserverrequest.h`

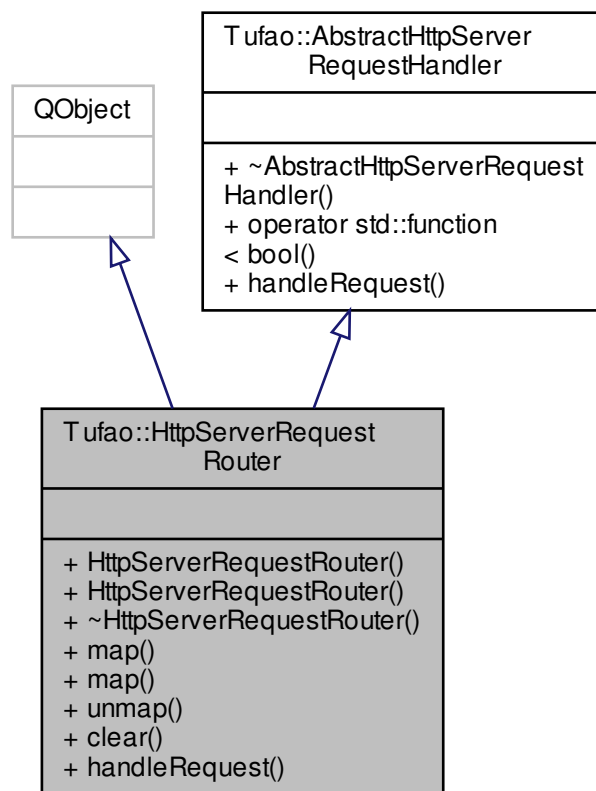
10.13 Tufao::HttpServerRequestRouter Class Reference

This class provides a robust and high performance HTTP request router.

Inheritance diagram for Tufao::HttpServerRequestRouter:



Collaboration diagram for Tufao::HttpServerRequestRouter:



Classes

- struct [Mapping](#)

This class describes a request handler and a filter.

Public Types

- typedef std::function< bool([HttpServerRequest](#) &, [HttpServerResponse](#) &)> [Handler](#)

It's a simple typedef for the type of handler accepted by the [HttpServerRequestRouter](#).

Public Slots

- bool [handleRequest](#) ([Tufao::HttpServerRequest](#) &request, [Tufao::HttpServerResponse](#) &response) override
It will route the request to the right handler.

Public Member Functions

- [HttpServerRequestRouter](#) (QObject *parent=0)
Constructs a [HttpServerRequestRouter](#) object.
- [HttpServerRequestRouter](#) (std::initializer_list< [Mapping](#) > mappings, QObject *parent=0)
Constructs a [HttpServerRequestRouter](#) object initialized with mappings.
- [~HttpServerRequestRouter](#) ()
Destroys the object.
- int [map](#) ([Mapping](#) map)
Chain map to the list of handlers available to handle requests.
- int [map](#) (std::initializer_list< [Mapping](#) > map)
Chain map to the list of handlers available to handle requests.
- void [unmap](#) (int index)
Removes the mapping at index.
- void [clear](#) ()
Removes all mappings.

10.13.1 Detailed Description

This class provides a robust and high performance HTTP request router.

It allows register a chain of request handlers. This router uses mapping rules based on the url's path component and http method to determine the correct handlers.

The type of mapping rules used in this class provides a predictable behaviour that is simple to understand and allow the use of caching algorithms to improve the performance.

When the router finds one matching request handler, it will call it passing the request and response objects. If the found handler cannot handle the request (this is indicated by the return value), the router will continue its quest in the search of a worthy handler. If the router fails in its quest (when no handlers are found or when none of the found handlers are able to respond the request), the `handleRequest` method in the router returns false and the connection remains open. This mean that you should always create a handler that responds to any request with a *404 not found* as the last handler in the most top-level request router.

The code below provides an example usage:

```
#include <QCoreApplication>

#include <Tufao/HttpPluginServer>
#include <Tufao/HttpFileServer>
#include <Tufao/NotFoundHandler>

#include <Tufao/HttpServerRequestRouter>
#include <Tufao/HttpServer>

using namespace Tufao;

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    HttpPluginServer plugins{"routes.json"};

    HttpServerRequestRouter router{
        {QRegularExpression{"", plugins},
        {QRegularExpression{"", HttpFileServer::handler("public")},
        {QRegularExpression{"", NotFoundHandler::handler()}}
    };

    HttpServer server;

    QObject::connect(&server, &HttpServer::requestReady,
                    &router, &HttpServerRequestRouter::handleRequest
    );

    server.listen(QHostAddress::Any, 8080);

    return a.exec();
}
```

Since

0.3

10.13.2 Member Typedef Documentation

10.13.2.1 Handler

```
typedef std::function<bool(HttpServerRequest&, HttpServerResponse&)> Tufao::HttpServerRequestRouter::Handler
```

It's a simple typedef for the type of handler accepted by the [HttpServerRequestRouter](#).

Since

1.0

10.13.3 Constructor & Destructor Documentation

10.13.3.1 HttpServerRequestRouter()

```
Tufao::HttpServerRequestRouter::HttpServerRequestRouter (  
    std::initializer_list< Mapping > mappings,  
    QObject * parent = 0 ) [explicit]
```

Constructs a [HttpServerRequestRouter](#) object initialized with mappings.

Since

1.0

10.13.4 Member Function Documentation

10.13.4.1 `handleRequest`

```
bool Tufao::HttpServerRequestRouter::handleRequest (
    Tufao::HttpServerRequest & request,
    Tufao::HttpServerResponse & response ) [override], [slot]
```

It will route the request to the right handler.

The handler will have access to the list of captured texts by the regular expression using [HttpServerRequest::customData](#).

Note

The router won't touch the [HttpServerRequest::customData](#) if the regex don't capture any text.

See example below:

```
bool Handler::handleRequest (HttpServerRequest &request,
                             HttpServerResponse &response)
{
    // ...

    QStringList args = request.customData().toMap()["args"].toStringList();

    // ...
}
```

If there is already an object set for this request, the router will do the following steps:

1. If the object is not a `QVariantMap`, override it
2. If the object already has a item with the key "args", but the value is not a `QStringList`, override the item
3. Append the list of captured texts in the object["args"]

Note

The request's custom data is copied at the beginning and is used to restore the custom data state before call every handler. The state will also be restored if no handlers are capable of handle the request.

Returns

Returns true if one handler able to respond the request is found.

Since

1.0

10.13.4.2 map() [1/2]

```
int Tufao::HttpServerRequestRouter::map (
    Mapping map )
```

Chain map to the list of handlers available to handle requests.

Returns

The index of the new mapping.

Since

1.0

10.13.4.3 map() [2/2]

```
int Tufao::HttpServerRequestRouter::map (
    std::initializer_list< Mapping > map )
```

Chain map to the list of handlers available to handle requests.

Returns

The index of the first element in the new mapping range.

Since

1.0

10.13.4.4 unmap()

```
void Tufao::HttpServerRequestRouter::unmap (
    int index )
```

Removes the mapping at index.

Since

1.0

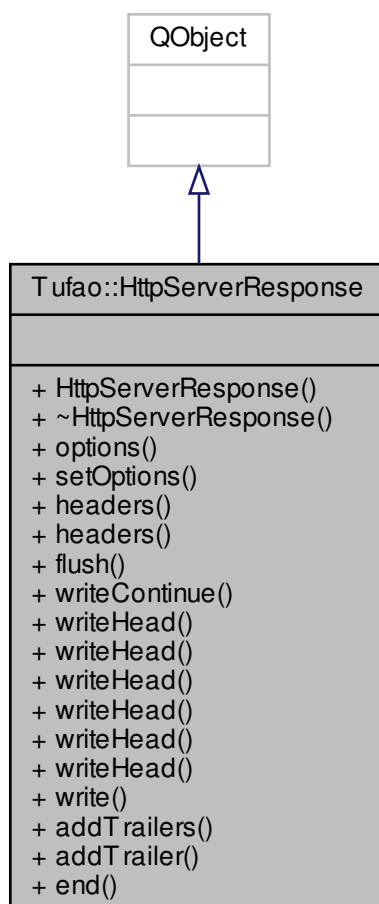
The documentation for this class was generated from the following file:

- httpserverrequestrouter.h

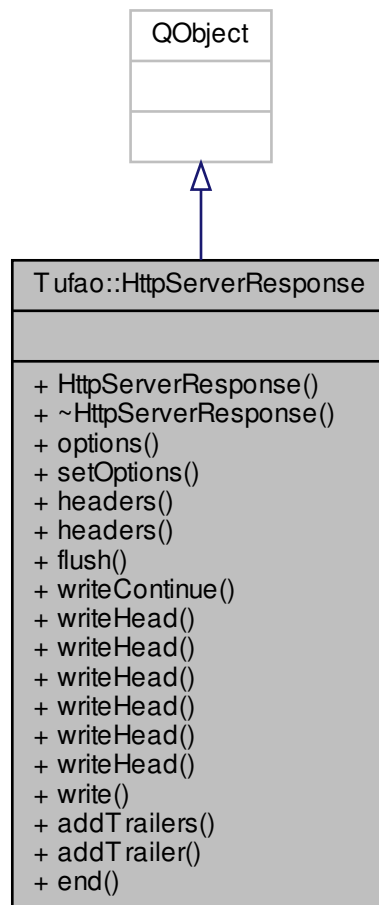
10.14 Tufao::HttpServerResponse Class Reference

The [Tufao::HttpServerResponse](#) is used to respond to a [Tufao::HttpServerRequest](#).

Inheritance diagram for Tufao::HttpServerResponse:



Collaboration diagram for Tufao::HttpServerResponse:



Public Types

- enum `Option` { `HTTP_1_0` = 1, `HTTP_1_1` = 1 << 1, `KEEP_ALIVE` = 1 << 2 }

This enum represents some aspects of a HTTP response.

Public Slots

- bool `writeContinue` ()
Sends a HTTP/1.1 100 Continue message to the client, indicating that the request body should be sent.
- bool `writeHead` (int statusCode, const QByteArray &reasonPhrase, const Headers &headers)
Sends a response header to the request.
- bool `writeHead` (int statusCode, const QByteArray &reasonPhrase)
This is an overloaded function.
- bool `writeHead` (HttpResponseStatus statusCode, const QByteArray &reasonPhrase, const Headers &headers)

- This is an overloaded function.*
- bool [writeHead](#) ([HttpResponseStatus](#) statusCode, const QByteArray &reasonPhrase)
This is an overloaded function.
- bool [writeHead](#) ([HttpResponseStatus](#) statusCode, const [Headers](#) &headers)
This is an overloaded function.
- bool [writeHead](#) ([HttpResponseStatus](#) statusCode)
This is an overloaded function.
- bool [write](#) (const QByteArray &chunk)
This sends a chunk of the response body.
- bool [addTrailers](#) (const [Headers](#) &headers)
This method adds HTTP trailing headers (a header but at the end of the message) to the response.
- bool [addTrailer](#) (const QByteArray &headerName, const QByteArray &headerValue)
This method adds one HTTP trailing header (a header but at the end of the message) to the response.
- bool [end](#) (const QByteArray &chunk=QByteArray())
This method signals to the server that all of the response headers and body has been sent; that server should consider this message complete.

Signals

- void [finished](#) ()
This signal is emitted when all bytes from the HTTP response message are written in the device/socket (not confuse with delivered).

Public Member Functions

- [HttpServerResponse](#) (QIODevice &device, Options [options](#)=Options(), QObject *parent=0)
Constructs a [Tufao::HttpServerResponse](#) object.
- [~HttpServerResponse](#) ()
Destroys the object.
- Options [options](#) () const
Returns the options passed to the object constructor.
- bool [setOptions](#) (Options [options](#))
Change the formatting options to [options](#).
- const [Headers](#) & [headers](#) () const
Returns a const reference to the headers which will be sent when the first piece of body is written.
- [Headers](#) & [headers](#) ()
Returns a reference to the headers which will be sent when the first piece of body is written.
- bool [flush](#) ()
This method calls [QAbstractSocket::flush](#) for the object passed in the constructor.

10.14.1 Detailed Description

The [Tufao::HttpServerResponse](#) is used to respond to a [Tufao::HttpServerRequest](#).

A response is built of well defined parts and must be sent ordered. The order to send these parts are:

- Status line: [Tufao::HttpServerResponse::writeHead](#)
- [Headers](#): Set them with [Tufao::HttpServerResponse::headers](#) and they will be automatically flushed when the first piece of body is written.
- Message body: Use [Tufao::HttpServerResponse::write](#) or [Tufao::HttpServerResponse::end](#)
- Trailers (optional): Use [Tufao::HttpServerResponse::addTrailers](#)
- EOF: Use [Tufao::HttpServerResponse::end](#)

Note

In HTTP/1.0 connections, it's not possible to send the message body in chunks and you must use [Tufao::HttpServerResponse::end](#) to send the full body at once. Additionally, HTTP/1.0 connections don't support trailers too.

See also

[Tufao::HttpServer](#)

10.14.2 Member Enumeration Documentation**10.14.2.1 Option**

```
enum Tufao::HttpServerResponse::Option
```

This enum represents some aspects of a HTTP response.

Enumerator

HTTP_1_0	A HTTP/1.0 response.
HTTP_1_1	A HTTP/1.1 response.
KEEP_ALIVE	The connection should use a persistent stream. Note Only supported in HTTP/1.1 connections.

10.14.3 Constructor & Destructor Documentation**10.14.3.1 HttpServerResponse()**

```
Tufao::HttpServerResponse::HttpServerResponse (
    QIODevice & device,
    Options options = Options(),
    QObject * parent = 0 ) [explicit]
```

Constructs a [Tufao::HttpServerResponse](#) object.

`parent` is passed to the `QObject` constructor.

Parameters

<i>options</i>	It controls some aspects of the response.
<i>device</i>	The socket used by Tufao::HttpServerResponse to write a HTTP response message.

Note

if `options` doesn't contain `HTTP_1_0` or `HTTP_1_1` set, the behaviour is undefined. If you set both flags, the behaviour is undefined also.

Since

1.0

10.14.4 Member Function Documentation**10.14.4.1 addTrailer**

```
bool Tufao::HttpServerResponse::addTrailer (
    const QByteArray & headerName,
    const QByteArray & headerValue ) [slot]
```

This method adds one HTTP trailing header (a header but at the end of the message) to the response.

Warning

Trailers will only be emitted if chunked encoding is used for the response; if it is not (e.g., if the request was sent by a HTTP/1.0 user agent), they will be silently discarded.

Note

A server **MUST NOT** use the trailer for any header fields unless at least one of the following is true:

- the request included a TE header field that indicates “trailers” is acceptable in the transfer-coding of the response;
- the server is the origin server (your server is not a proxy or a tunnel) for the response, the trailer fields consist entirely of optional metadata, and the recipient could use the message (in a manner acceptable to the origin server) without receiving this metadata. In other words, the origin server is willing to accept the possibility that the trailer fields might be silently discarded along the path to the client.

See also

[Tufao::HttpServerResponse::addTrailers](#)

10.14.4.2 addTrailers

```
bool Tufao::HttpServerResponse::addTrailers (
    const Headers & headers ) [slot]
```

This method adds HTTP trailing headers (a header but at the end of the message) to the response.

Warning

Trailers will only be emitted if chunked encoding is used for the response; if it is not (e.g., if the request was sent by a HTTP/1.0 user agent), they will be silently discarded.

Note

A server MUST NOT use the trailer for any header fields unless at least one of the following is true:

- the request included a TE header field that indicates “trailers” is acceptable in the transfer-coding of the response;
- the server is the origin server (your server is not a proxy or a tunnel) for the response, the trailer fields consist entirely of optional metadata, and the recipient could use the message (in a manner acceptable to the origin server) without receiving this metadata. In other words, the origin server is willing to accept the possibility that the trailer fields might be silently discarded along the path to the client.

See also

[Tufao::HttpServerResponse::addTrailer](#)

10.14.4.3 end

```
bool Tufao::HttpServerResponse::end (
    const QByteArray & chunk = QByteArray() ) [slot]
```

This method signals to the server that all of the response headers and body has been sent; that server should consider this message complete.

It MUST be called on each response exactly one time.

Parameters

<i>chunk</i>	If specified, it is equivalent to calling Tufao::HttpServerResponse::write followed by Tufao::HttpServerResponse::end
--------------	---

10.14.4.4 finished

```
void Tufao::HttpServerResponse::finished ( ) [signal]
```

This signal is emitted when all bytes from the HTTP response message are written in the device/socket (not confuse with delivered).

Call `Tufao::HttpServerResponse::end` will cause this signal to be emitted.

10.14.4.5 flush()

```
bool Tufao::HttpServerResponse::flush ( )
```

This method calls `QAbstractSocket::flush` for the object passed in the constructor.

It'll do nothing and return false if the object passed in the constructor isn't an actual `QAbstractSocket` instance.

Since

0.3

10.14.4.6 headers() [1/2]

```
const Headers& Tufao::HttpServerResponse::headers ( ) const
```

Returns a const reference to the headers which will be sent when the first piece of body is written.

Since

0.4

10.14.4.7 headers() [2/2]

```
Headers& Tufao::HttpServerResponse::headers ( )
```

Returns a reference to the headers which will be sent when the first piece of body is written.

Use this reference to send custom headers.

Note

Change this object when the first piece of the message body was already written won't take any effects. However the object will retain the changes.

10.14.4.8 setOptions()

```
bool Tufao::HttpServerResponse::setOptions (
    Options options )
```

Change the formatting options to `options`.

Note

if `options` doesn't contain `HTTP_1_0` or `HTTP_1_1` set, the behaviour is undefined. If you set both flags, the behaviour is undefined also.

Returns

true if successful.
false if wasn't possible to change options. This can happen if the first chunk of data was already sent.

Since

1.0

10.14.4.9 write

```
bool Tufao::HttpServerResponse::write (
    const QByteArray & chunk ) [slot]
```

This sends a chunk of the response body.

This method may be called multiple times to provide successive parts of the body.

Note

This is the raw HTTP body and has nothing to do with higher-level multi-part body encodings that may be used.

The first time [Tufao::HttpServerResponse::write](#) is called, it will send the buffered headers and the first body chunk to the client. The second time [Tufao::HttpServerResponse::write](#) is called, it assumes you're going to streaming data, and sends that separately. That is, the response is buffered up to the first chunk of body.

If you call this function with a empty byte array, it will do nothing.

Note

HTTP/1.0 user agents don't support chunked entities. To overcome this limitation, [Tufao::HttpServerResponse](#) will buffer the chunks and send the full message at once. If you want a performance boost, treat HTTP/1.0 clients differently.

See also

[operator<<\(HttpServerResponse&,const QByteArray&\)](#)

10.14.4.10 writeContinue

```
bool Tufao::HttpServerResponse::writeContinue ( ) [slot]
```

Sends a *HTTP/1.1 100 Continue* message to the client, indicating that the request body should be sent.

You should write a *HTTP/1.1 100 Continue* response to requests that include *100-continue* in the header *Expect* if you are willing to accept the request body based on the headers sent.

If you don't want to accept the request body, you should respond it with an `Tufao::HttpServerResponse::EXPECTATION_FAILED` response status.

The purpose of the *100 Continue* status is to decrease the network traffic by avoiding the transfer of data that the server would reject anyway.

Warning

It's not possible to send a *HTTP/1.1 100 Continue* to HTTP/1.0 clients.

10.14.4.11 writeHead [1/6]

```
bool Tufao::HttpServerResponse::writeHead (
    int statusCode,
    const QByteArray & reasonPhrase,
    const Headers & headers ) [slot]
```

Sends a response header to the request.

Call this function after the first chunk of entity body data was already sent (calling `Tufao::HttpServerResponse::write`) will have no effect.

Parameters

<i>statusCode</i>	The status code is a 3-digit HTTP status code.
<i>reasonPhrase</i>	A human-readable reasonPhrase.
<i>headers</i>	The response headers.

10.14.4.12 writeHead [2/6]

```
bool Tufao::HttpServerResponse::writeHead (
    int statusCode,
    const QByteArray & reasonPhrase ) [slot]
```

This is an overloaded function.

See also

[Tufao::HttpServerResponse::writeHead\(int, const QByteArray&, const Headers&\)](#)

10.14.4.13 writeHead [3/6]

```
bool Tufao::HttpServerResponse::writeHead (
    HttpResponseStatus statusCode,
    const QByteArray & reasonPhrase,
    const Headers & headers ) [slot]
```

This is an overloaded function.

See also

[Tufao::HttpServerResponse::writeHead\(int, const QByteArray&, const Headers&\)](#)

10.14.4.14 writeHead [4/6]

```
bool Tufao::HttpServerResponse::writeHead (
    HttpResponseStatus statusCode,
    const QByteArray & reasonPhrase ) [slot]
```

This is an overloaded function.

See also

[Tufao::HttpServerResponse::writeHead\(int, const QByteArray&, const Headers&\)](#)

10.14.4.15 writeHead [5/6]

```
bool Tufao::HttpServerResponse::writeHead (
    HttpResponseStatus statusCode,
    const Headers & headers ) [slot]
```

This is an overloaded function.

See also

[Tufao::HttpServerResponse::writeHead\(int, const QByteArray&, const Headers&\)](#)

10.14.4.16 writeHead [6/6]

```
bool Tufao::HttpServerResponse::writeHead (
    HttpResponseStatus statusCode ) [slot]
```

This is an overloaded function.

See also

[Tufao::HttpServerResponse::writeHead\(int, const QByteArray&, const Headers&\)](#)

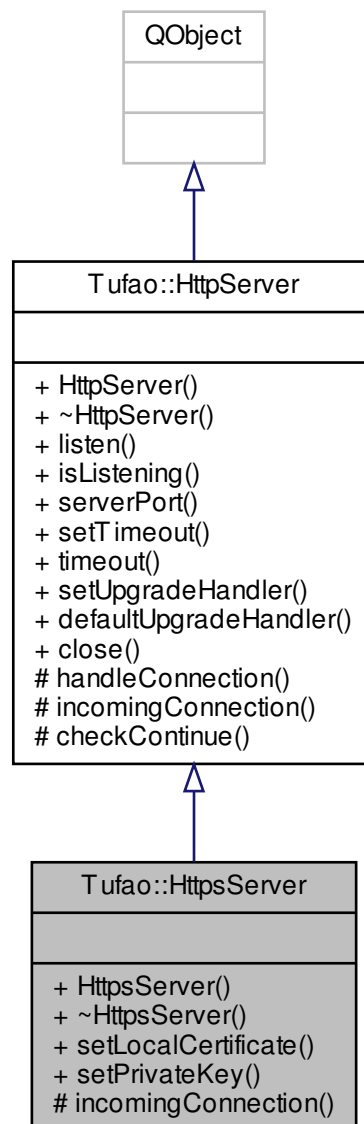
The documentation for this class was generated from the following file:

- [httpserverresponse.h](#)

10.15 Tufao::HttpsServer Class Reference

[Tufao::HttpsServer](#) is a subclass of [Tufao::HttpServer](#) that provides secure communication with the client.

Inheritance diagram for Tufao::HttpsServer:



Collaboration diagram for Tufao::HttpsServer:



Public Member Functions

- **HttpsServer** (QObject *parent=0)
- `~HttpsServer` ()
Destroys the object.
- void `setLocalCertificate` (const QSslCertificate &certificate)
Sets the local certificate to `certificate`.
- void `setPrivateKey` (const QSslKey &key)
Sets the private key to `key`.

Protected Member Functions

- void **incomingConnection** (qintptr socketDescriptor) override

Additional Inherited Members

10.15.1 Detailed Description

[Tufao::HttpsServer](#) is a subclass of [Tufao::HttpServer](#) that provides secure communication with the client.

It does this using socket streams over TLS connections.

This combination (HTTP + SSL/TLS) is know as HTTP Secure and provides encrypted communication.

To use HTTPS in Tufão, just set the local certificate and private key before call [Tufao::HttpsServer::listen](#). The default port for this protocol is 443.

Note

You should also pay to a trusted certificate authority to sign your certificate if you are willing to provide secure identification also.

The use of HTTPS implies an extra overhead in the software, limiting the number of requests that can be served per time and should be moderated. It's common to use it only in pages that handles more sensitive information, such as login pages and payment transactions.

See also

[HttpsServer::setLocalCertificate](#) [HttpsServer::setPrivateKey](#)

10.15.2 Member Function Documentation

10.15.2.1 setLocalCertificate()

```
void Tufao::HttpsServer::setLocalCertificate (
    const QSslCertificate & certificate )
```

Sets the local certificate to `certificate`.

Note

This member function should be called before [Tufao::HttpsServer::listen](#)

10.15.2.2 setPrivateKey()

```
void Tufao::HttpsServer::setPrivateKey (
    const QSslKey & key )
```

Sets the private key to key.

Warning

Remember that encryption security relies on the fact that no one knows your private key.

Note

This member function should be called before [Tufao::HttpsServer::listen](#)

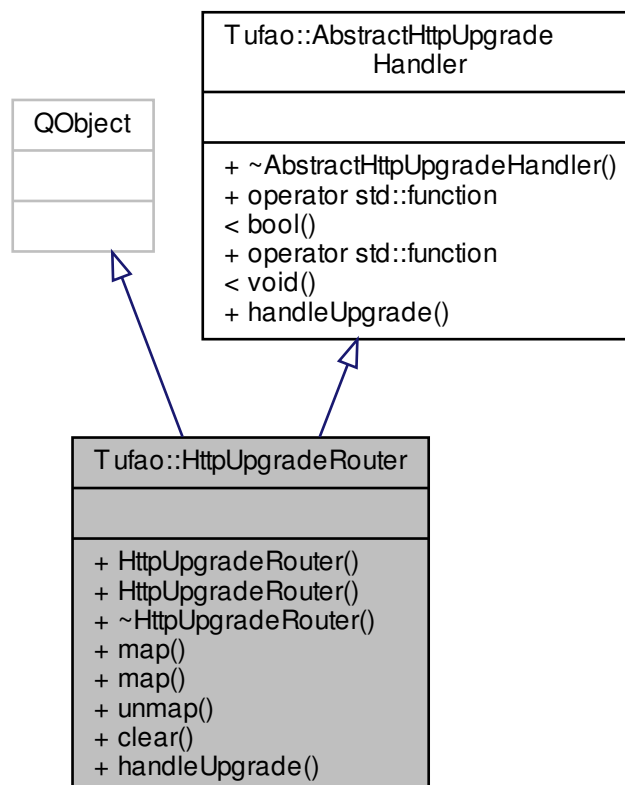
The documentation for this class was generated from the following file:

- httpsserver.h

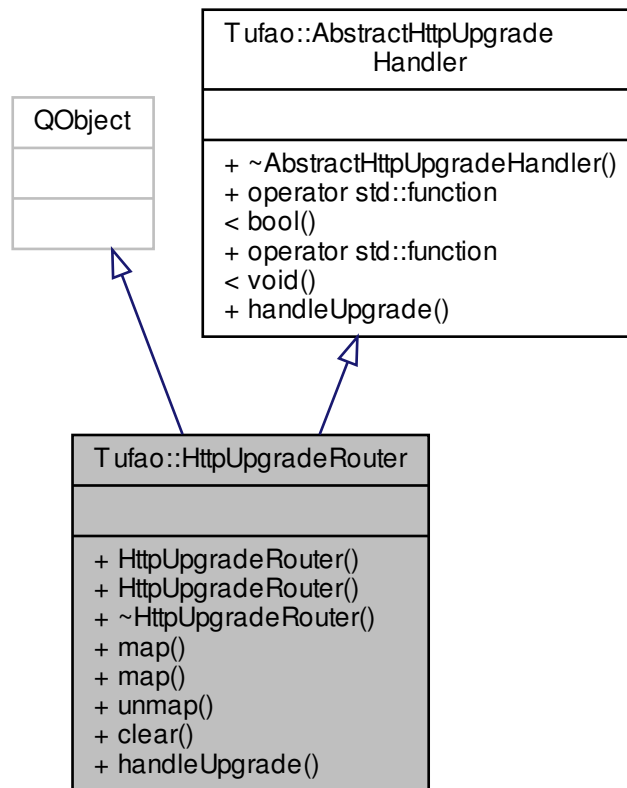
10.16 Tufao::HttpUpgradeRouter Class Reference

This class provides a robust and high performance HTTP request router.

Inheritance diagram for Tufao::HttpUpgradeRouter:



Collaboration diagram for Tufao::HttpUpgradeRouter:



Classes

- struct [Mapping](#)

This class describes a request handler and a filter.

Public Types

- typedef std::function< bool([HttpServerRequest](#) &, const QByteArray &head)> [Handler](#)

It's a simple typedef for the type of handler accepted by the [HttpUpgradeRouter](#).

Public Slots

- bool [handleUpgrade](#) ([Tufao::HttpServerRequest](#) &request, const QByteArray &head) override

It will route the request to the right handler.

Public Member Functions

- [HttpUpgradeRouter](#) (QObject *parent=0)
Constructs a [HttpServerRequestRouter](#) object.
- [HttpUpgradeRouter](#) (std::initializer_list< [Mapping](#) > mappings, QObject *parent=0)
Constructs a [HttpServerRequestRouter](#) object initialized with mappings.
- [~HttpUpgradeRouter](#) ()
Destroys the object.
- int [map](#) ([Mapping](#) map)
Chain map to the list of handlers available to handle requests.
- int [map](#) (std::initializer_list< [Mapping](#) > map)
Chain map to the list of handlers available to handle requests.
- void [unmap](#) (int index)
Removes the mapping at index.
- void [clear](#) ()
Removes all mappings.

10.16.1 Detailed Description

This class provides a robust and high performance HTTP request router.

It allows to register a chain of upgrade handlers. This router uses mapping rules based on the url's path component to determine the correct handlers.

The type of mapping rules used in this class provides a predictable behaviour that is simple to understand and allow the use of caching algorithms to improve the performance.

When the router finds one matching request handler, it will call it. If the found handler cannot handle the request (this is indicated by the return value), the router will continue its quest in the search of a worthy handler. If the router fails in its quest (when no handlers are found or when none of the found handlers are able to respond the request), the `handleRequest` method in the router returns false and the connection remains open.

The code below provides an example usage:

```
int main(int argc, char *argv[])
{
    using namespace Tufao;

    QCoreApplication a(argc, argv);

    // ...

    HttpUpgradeRouter upgradeRouter{
        {QRegularExpression{"^/chat$"}, [] (HttpServerRequest &request,
                                           const QByteArray &head) -> bool {
            WebSocket *ws = new WebSocket;

            ws->startServerHandshake(request, head);

            // ...
        }}
    };

    HttpServer server;
    server.setUpgradeHandler(upgradeRouter);

    QObject::connect(&server, &HttpServer::requestReady,
                    &router, &HttpServerRequestRouter::handleRequest
    );

    server.listen(QHostAddress::Any, 8080);

    return a.exec();
}
```

See also

[HttpServer::setUpgradeHandler](#)

Since

1.0

10.16.2 Member Typedef Documentation

10.16.2.1 Handler

```
typedef std::function<bool(HttpServerRequest&, const QByteArray &head)> Tufao::HttpUpgradeRouter::Handler
```

It's a simple typedef for the type of handler accepted by the [HttpUpgradeRouter](#).

Since

1.0

10.16.3 Member Function Documentation

10.16.3.1 handleUpgrade

```
bool Tufao::HttpUpgradeRouter::handleUpgrade (  
    Tufao::HttpServerRequest & request,  
    const QByteArray & head ) [override], [slot]
```

It will route the request to the right handler.

The handler will have access to the list of captured texts by the regular expression using [HttpServerRequest::customData](#).

Note

The router won't touch the [HttpServerRequest::customData](#) if the regex don't capture any text.

See example below:

```
bool Handler::handleRequest(HttpServerRequest &request,  
                           HttpServerResponse &response)  
{  
    // ...  
  
    QStringList args = request.customData().toMap()["args"].toStringList();  
  
    // ...  
}
```

If there is already an object set for this request, the router will do the following steps:

1. If the object is not a [QVariantMap](#), override it
2. If the object already has a item with the key "args", but the value is not a [QStringList](#), override the item
3. Append the list of captured texts in the object["args"]

Note

The request's custom data is copied at the beginning and is used to restore the custom data state before call every handler. The state will also be restored if no handlers are capable of handle the request.

Returns

Returns true if one handler able to respond the request is found.

Since

1.0

10.16.3.2 map() [1/2]

```
int Tufao::HttpUpgradeRouter::map (
    Mapping map )
```

Chain map to the list of handlers available to handle requests.

Returns

The index of the new mapping.

10.16.3.3 map() [2/2]

```
int Tufao::HttpUpgradeRouter::map (
    std::initializer_list< Mapping > map )
```

Chain map to the list of handlers available to handle requests.

Returns

The index of the first element in the new mapping range.

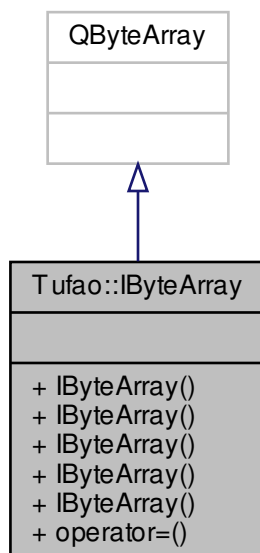
The documentation for this class was generated from the following file:

- httpupgraderouter.h

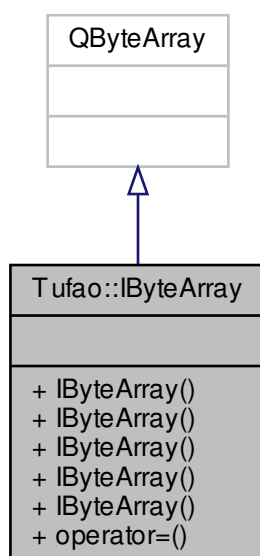
10.17 Tufao::IByteArray Class Reference

This class provides a case insensitive QByteArray.

Inheritance diagram for Tufao::IByteArray:



Collaboration diagram for Tufao::IByteArray:



Public Member Functions

- **IByteArray** (const QByteArray &ba)
- **IByteArray** (const char *str)
- **IByteArray** (const char *data, int size)
- **IByteArray** (int size, char ch)
- **IByteArray** & **operator=** (const QByteArray &ba)

10.17.1 Detailed Description

This class provides a case insensitive QByteArray.

It inherits from QByteArray and provides non-member functions to overload the common operators:

- operator !=
- operator <
- operator <=
- operator ==
- operator >
- operator >=

Note

Use of overloaded operator '<' is intentionally ambiguous when you combine [IByteArray](#) and const char *. This design forces you to make your intent explicit using explicit casts.

Note

All member functions of this class are inlined and should add the minimum (if any) of overhead.

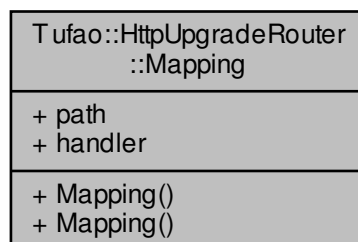
The documentation for this class was generated from the following file:

- ibytearray.h

10.18 Tufao::HttpUpgradeRouter::Mapping Struct Reference

This class describes a request handler and a filter.

Collaboration diagram for Tufao::HttpUpgradeRouter::Mapping:



Public Member Functions

- [Mapping](#) (QRegularExpression [path](#), [Handler](#) handler)
Constructs a [Mapping](#) object using `path` as filter and `handler` as handler.
- [Mapping](#) ()=default
Constructs an empty [Mapping](#) object.

Public Attributes

- QRegularExpression [path](#)
This attribute is used to filter requests based on the url's path component.
- [Handler](#) **handler**

10.18.1 Detailed Description

This class describes a request handler and a filter.

The filter is very basic and only can select requests based on the url's path component.

The handlers return a boolean indicating whether it's able to handle the request. You can use this return value to overcome the simplistic nature of the provided filter.

10.18.2 Member Data Documentation

10.18.2.1 `path`

```
QRegularExpression Tufao::HttpUpgradeRouter::Mapping::path
```

This attribute is used to filter requests based on the url's path component.

It's a regular expression, so you have powerful control.

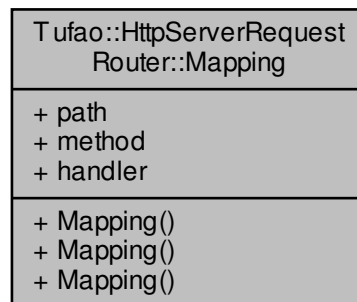
The documentation for this struct was generated from the following file:

- `httpupgraderouter.h`

10.19 Tufao::HttpServerRequestRouter::Mapping Struct Reference

This class describes a request handler and a filter.

Collaboration diagram for Tufao::HttpServerRequestRouter::Mapping:



Public Member Functions

- [Mapping](#) (QRegularExpression [path](#), [Handler](#) handler)
Constructs a [Mapping](#) object using [path](#) as filter and [handler](#) as handler.
- [Mapping](#) (QRegularExpression [path](#), QByteArray [method](#), [Handler](#) handler)
Constructs a [Mapping](#) object using [path](#) and [method](#) as filters and [handler](#) as handler.
- [Mapping](#) ()=default
Constructs an empty [Mapping](#) object.

Public Attributes

- QRegularExpression [path](#)
This attribute is used to filter requests based on the url's path component.
- QByteArray [method](#)
This attribute is used to filter requests based on the HTTP method.
- [Handler](#) **handler**

10.19.1 Detailed Description

This class describes a request handler and a filter.

The filter is very basic and only can select requests based on the url's path component and, optionally, the http method.

The handlers return a boolean indicating whether it's able to handle the request. You can use this return value to overcome the simplistic nature of the provided filter or to respond requests partially (like fill headers that must be present in every response).

Since

1.0

10.19.2 Member Data Documentation

10.19.2.1 method

```
QByteArray Tufao::HttpServerRequestRouter::Mapping::method
```

This attribute is used to filter requests based on the HTTP method.

The filter should work by simply comparing strings.

Note

If this attribute is left null, it won't be used by the filter.

10.19.2.2 path

```
QRegularExpression Tufao::HttpServerRequestRouter::Mapping::path
```

This attribute is used to filter requests based on the url's path component.

It's a regular expression, so you have powerful control.

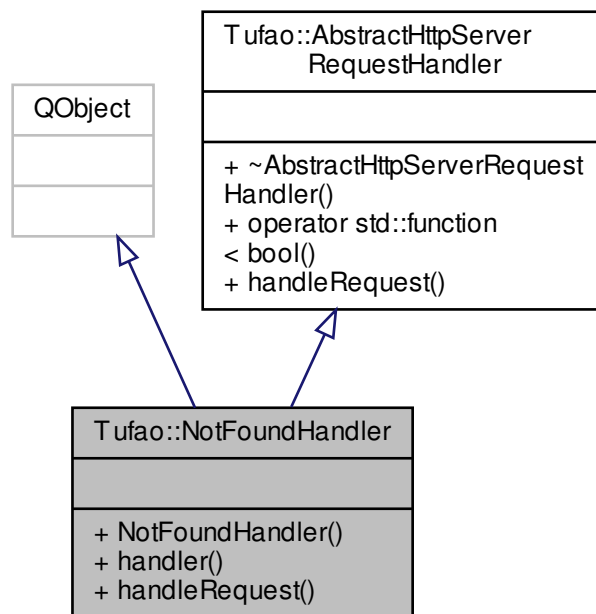
The documentation for this struct was generated from the following file:

- httpserverrequestrouter.h

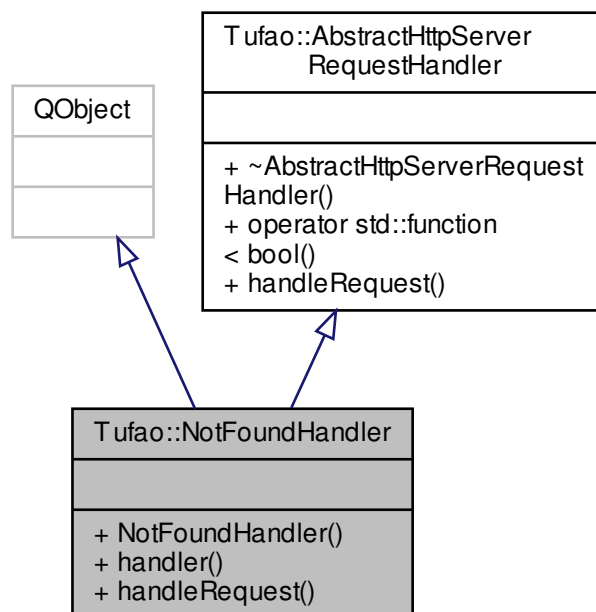
10.20 Tufao::NotFoundHandler Class Reference

A handler that responds with "Not found" to every request.

Inheritance diagram for Tufao::NotFoundHandler:



Collaboration diagram for Tufao::NotFoundHandler:



Public Slots

- bool [handleRequest](#) ([Tufao::HttpRequest](#) &request, [Tufao::HttpServerResponse](#) &response) override
It responds to the request with a not found message.

Public Member Functions

- [NotFoundHandler](#) (QObject *parent=0)
Constructs a [NotFoundHandler](#) object.

Static Public Member Functions

- static std::function< bool([HttpRequest](#) &, [HttpServerResponse](#) &)> [handler](#) ()
Returns a handler that don't depends on another object.

10.20.1 Detailed Description

A handler that responds with "Not found" to every request.

Its purpose is to avoid boilerplate code.

Since

1.0

10.20.2 Constructor & Destructor Documentation

10.20.2.1 [NotFoundHandler](#)()

```
Tufao::NotFoundHandler::NotFoundHandler (  
    QObject * parent = 0 ) [explicit]
```

Constructs a [NotFoundHandler](#) object.

`parent` is passed to the QObject constructor.

10.20.3 Member Function Documentation

10.20.3.1 handler()

```
std::function< bool(HttpRequest &, HttpResponse &)> Tufao::NotFoundHandler::handler
( ) [inline], [static]
```

Returns a handler that don't depends on another object.

The purpose of this alternative handler is to free you of the worry of maintain the [NotFoundHandler](#)'s object (lifetime) while the functor object is being used.

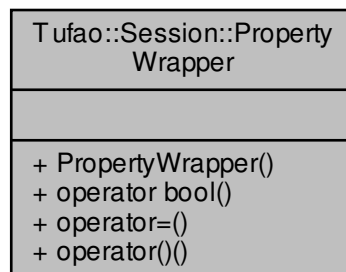
The documentation for this class was generated from the following file:

- notfoundhandler.h

10.21 Tufao::Session::PropertyWrapper Class Reference

Provides a object that give less verbose access to a session property.

Collaboration diagram for Tufao::Session::PropertyWrapper:



Public Member Functions

- [PropertyWrapper](#) ([Session](#) &session, const QByteArray &key)
Constructs a new PropertyWrape object.
- [operator bool](#) () const
Returns true if this property exists for this session.
- [PropertyWrapper](#) & [operator=](#) (const QVariant &value)
Assigns value to this property and returns a reference to itself.
- QVariant [operator\(\)](#) ()
Returns this property's value.

10.21.1 Detailed Description

Provides a object that give less verbose access to a session property.

Note

You shouldn't create it directly, but use [Session::operator \[\]](#).

See also

[Session](#)

10.21.2 Constructor & Destructor Documentation

10.21.2.1 PropertyWrapper()

```
Tufao::Session::PropertyWrapper::PropertyWrapper (
    Session & session,
    const QByteArray & key ) [inline]
```

Constructs a new PropertyWrappe object.

Note

This object will use `session` in every operation and you should ensure that `session` isn't destructed before the [PropertyWrapper](#) object.

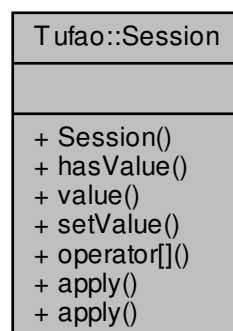
The documentation for this class was generated from the following file:

- session.h

10.22 Tufao::Session Class Reference

This class provides easier access to the session's properties.

Collaboration diagram for Tufao::Session:



Classes

- class [PropertyWrapper](#)

Provides a object that give less verbose access to a session property.

Public Member Functions

- [Session](#) ([SessionStore](#) &store, const [HttpServerRequest](#) &request, [HttpServerResponse](#) &response)
Constructs a new [Session](#) object.
- bool [hasValue](#) (const QByteArray &key) const
*Returns true if the session has a property accessible through *key*.*
- QVariant [value](#) (const QByteArray &key) const
*Returns the value of the property referenced by *key*, or a null QVariant if the property isn't found.*
- void [setValue](#) (const QByteArray &key, const QVariant &value)
*Sets the property's value referenced by *key* to *value*.*
- [PropertyWrapper operator\[\]](#) (const QByteArray &key)
*Returns a [PropertyWrapper](#) that will remember the *key* used to manipulate the session property.*

Static Public Member Functions

- template<class F >
static void [apply](#) ([SessionStore](#) &store, const QByteArray &property, const [HttpServerRequest](#) &request, [HttpServerResponse](#) &response, F f)
Takes a functor to access a session's property.
- template<class F >
static void [apply](#) ([SessionStore](#) &store, const [HttpServerRequest](#) &request, [HttpServerResponse](#) &response, F f)
Takes a functor to access the session's properties.

10.22.1 Detailed Description

This class provides easier access to the session's properties.

It uses C++ features used in containers to provide a familiar interface, such as overloading the operator [].

```
bool RequestHandler::handleRequest(Tufao::HttpServerRequest &request,
                                   Tufao::HttpServerResponse &response)
{
    Tufao::Session s(store, request, response);

    s["access"] = s["access"]().toInt() + 1;

    response.writeHead(200, "OK");

    response << "You have "
              << QByteArray::number(s["access"].toInt())
              << " access";

    response.end();
    return true;
}
```

Note

All member functions of this class are inlined and should add the minimum (if any) of overhead.

See also

[SessionStore](#)

Since

0.4

10.22.2 Constructor & Destructor Documentation

10.22.2.1 Session()

```
Tufao::Session::Session (
    SessionStore & store,
    const HttpServerRequest & request,
    HttpServerResponse & response ) [inline]
```

Constructs a new [Session](#) object.

The object will use `store`, `request` and `response` in every operation and you should ensure these objects aren't destructed before this [Session](#) object.

10.22.3 Member Function Documentation

10.22.3.1 apply() [1/2]

```
template<class F >
static void Tufao::Session::apply (
    SessionStore & store,
    const QByteArray & property,
    const HttpServerRequest & request,
    HttpServerResponse & response,
    F f ) [inline], [static]
```

Takes a functor to access a session's property.

Parameters

<code>f</code>	a functor that receives a QVariant object reference as an argument.
----------------	---

Note

After the functor returns, the property is updated.

```
bool RequestHandler::handleRequest(Tufao::HttpServerRequest &request,
                                   Tufao::HttpServerResponse &response)
{
    response.writeHead(200, "OK");

    Tufao::Session::apply(store, "access", request, response,
                          [&response](QVariant &access) {
                              access = access.toInt() + 1;

                              response << "You visited this page "
                                      << QByteArray::number(access.toInt())
                                      << " times";
                          });

    response.end();
    return true;
}
```

Since

1.0

10.22.3.2 apply() [2/2]

```
template<class F >
static void Tufao::Session::apply (
    SessionStore & store,
    const HttpServerRequest & request,
    HttpServerResponse & response,
    F f ) [inline], [static]
```

Takes a functor to access the session's properties.

Parameters

<i>f</i>	a functor that receives a QMap<QByteArray, QVariant> object reference as an argument.
----------	---

Note

After the functor returns, the property is updated.

```
bool RequestHandler::handleRequest(Tufao::HttpServerRequest &request,
                                   Tufao::HttpServerResponse &response)
{
    response.writeHead(200, "OK");

    Tufao::Session::apply(store, request, response,
        [&response](QMap<QByteArray, QVariant> &properties) {
            properties["access"] = properties["access"].toInt() + 1;

            response << "You visited this page "
                << QByteArray::number(properties["access"].toInt())
                << " times";
        });

    response.end();
    return true;
}
```

Note

If the session may contain a lot of properties and you aren't going to access most of them, this helper function might be inefficient.

Since

1.0

10.22.3.3 hasValue()

```
bool Tufao::Session::hasValue (
    const QByteArray & key ) const [inline]
```

Returns true if the session has a property accessible through `key`.

See also

[SessionStore::hasProperty](#)

10.22.3.4 setValue()

```
void Tufao::Session::setValue (
    const QByteArray & key,
    const QVariant & value ) [inline]
```

Sets the property's value referenced by `key` to `value`.

See also

[SessionStore::setProperty](#)

10.22.3.5 value()

```
QVariant Tufao::Session::value (
    const QByteArray & key ) const [inline]
```

Returns the value of the property referenced by `key`, or a null `QVariant` if the property isn't found.

See also

[SessionStore::property](#)

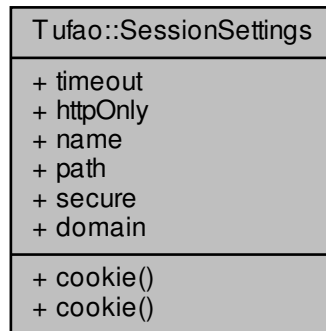
The documentation for this class was generated from the following file:

- `session.h`

10.23 Tufao::SessionSettings Struct Reference

The [SessionSettings](#) class exposes details that sessions use to handle cookies.

Collaboration diagram for Tufao::SessionSettings:



Public Member Functions

- QNetworkCookie [cookie](#) (const QByteArray &value=QByteArray()) const
Creates a cookie, using `value` as the cookie's value.

Static Public Member Functions

- static QNetworkCookie [cookie](#) (const [SessionSettings](#) &settings, const QByteArray &value=QByteArray())
Creates a cookie, using `value` as the cookie's value and `settings` as cookie's settings.

Public Attributes

- int [timeout](#)
Define the lifetime of cookies generated by this object (a timeout specified in minutes).
- bool [httpOnly](#)
Whether cookies generated by this object should only be used in HTTP requests.
- QByteArray [name](#)
The name to which cookies generated by this object are used.
- QByteArray [path](#)
The set of paths to which cookies generated by this object are used.
- bool [secure](#)
Whether cookies generated by this object should only be used through secure connections.
- QByteArray [domain](#)
The hosts to which cookies generated by this object are used.

10.23.1 Detailed Description

The [SessionSettings](#) class exposes details that sessions use to handle cookies.

Cookies are a mechanism to store state in the mostly stateless HTTP protocol. These details are the cookies attributes.

Note

Cookies don't provide isolation by port. For example, if a cookie is accessible by a service running on one port, it will also be accessible by a service running on another port on the same server.

Note

Cookies also don't provide isolation by scheme (HTTPS, HTTP, FTP, ...).

Warning

You should *not* create `SessionSetting` objects with equal names and different domain and paths hoping that [SessionStore](#) and some other objects making use of [SessionSettings](#) will work correctly. When a user agent sends a cookie, the only attributes sent in the request are the name and value pair, making impossible, in several cases, to identify the right cookie.

Since

0.4

10.23.2 Member Data Documentation

10.23.2.1 domain

```
QByteArray Tufao::SessionSettings::domain
```

The hosts to which cookies generated by this object are used.

Note

Subdomains are also considered. For example, if this value is "example.com", the cookie generated by this [SessionSettings](#) object will also be used when the user agent request some resource to the host "www.example.com" and "www.corp.example.com".

Note

User agents will reject cookies unless this attribute specifies a scope for the cookie that would include the origin server. For example, it will accept the value "example.com" or "foo.example.com" coming from the server "foo.example.com", but it will reject the value "bar.example.com".

For security reasons, many user agents are configured to reject values that correspond to public suffixes, such as "com" and "co.uk".

Note

If is not specified, the default behaviour, the user agent will only include the cookie to requests made to the origin server. In other words, it will, for example, exclude any subdomains.

10.23.2.2 httpOnly

```
bool Tufao::SessionSettings::httpOnly
```

Whether cookies generated by this object should only be used in HTTP requests.

It prevents, for example, scripting engines in the user agent from accessing the cookie.

Note

You should turn this attribute true if you will use this cookie to store sensitive data.

10.23.2.3 name

```
QByteArray Tufao::SessionSettings::name
```

The name to which cookies generated by this object are used.

This is the main cookie access key.

10.23.2.4 path

```
QByteArray Tufao::SessionSettings::path
```

The set of paths to which cookies generated by this object are used.

Let's name this value as `cookiePath` and the path component of the uri of a request as `requestPath`. The cookie will be included in a request if one of the following conditions is true:

- `cookiePath == requestPath`
- `requestPath.startsWith(cookiePath)`
- `requestPath[0] == '/' && requestPath.mid(1).startsWith(cookiePath)`

Note

If it's not specified, the user agent will choose a path based on the current request's uri path component.

Note

Cookies don't provide integrity protection to this attribute. For example, a service running on the path `"/foo"` can set a cookie with a path attribute with the value `"/bar"`. As a result, servers *should not* both run mutually distrusting services on different paths of the same host and use cookies to store sensitive data.

10.23.2.5 secure

```
bool Tufao::SessionSettings::secure
```

Whether cookies generated by this object should *only* be used through secure connections.

What "secure" channels means are defined by the user agent. This is typically HTTP over TLS.

Note

This attribute only protects cookie's confidentiality. An active network attacker can overwrite secure cookies from an insecure channel, disrupting their integrity.

You should turn this attribute true if you will use this cookie to store sensitive data.

See also

[HttpsServer](#)

10.23.2.6 timeout

```
int Tufao::SessionSettings::timeout
```

Define the lifetime of cookies generated by this object (a timeout specified in minutes).

The expiration date time is renewed every time SessionSetting generates a cookie.

Note

When not specified (the value is zero), the cookie expires at the end of the user agent current session (as defined by the user agent).

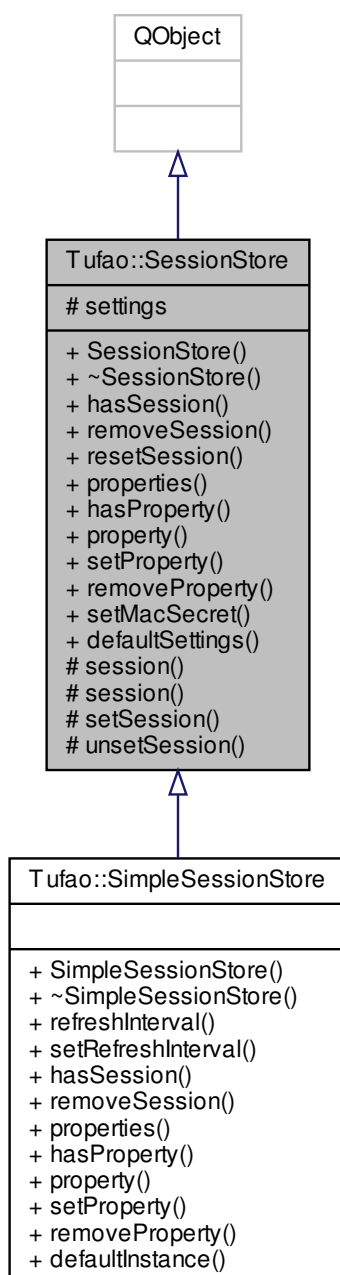
The documentation for this struct was generated from the following file:

- sessionsettings.h

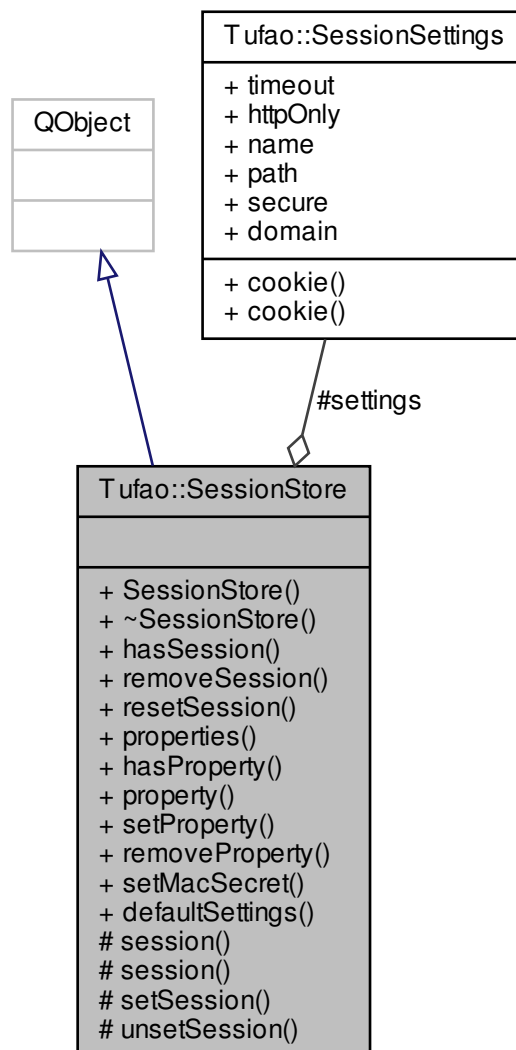
10.24 Tufao::SessionStore Class Reference

[SessionStore](#) class can be used to store data that must persist among different requests.

Inheritance diagram for Tufao::SessionStore:



Collaboration diagram for Tufao::SessionStore:



Public Member Functions

- [SessionStore](#) (const [SessionSettings](#) &settings=defaultSettings(), QObject *parent=0)
Constructs a [SessionStore](#) object.
- [~SessionStore](#) ()
Destructs a [SessionStore](#) object.
- virtual bool [hasSession](#) (const [HttpRequest](#) &request) const =0
*Returns true if a session has been set the *request*.*
- virtual void [removeSession](#) (const [HttpRequest](#) &request, [HttpServerResponse](#) &response)=0
*Removes the session, if any, in the *request*.*
- void [resetSession](#) ([HttpRequest](#) &request) const
*This method removes all cookies matching with this store's settings from *request*.*

- virtual QList< QByteArray > [properties](#) (const [HttpRequest](#) &request, const [HttpResponse](#) &response) const =0
Returns a list of set properties to this session.
- virtual bool [hasProperty](#) (const [HttpRequest](#) &request, const [HttpResponse](#) &response, const QByteArray &key) const =0
Returns true if the session has a property named *key*.
- virtual QVariant [property](#) (const [HttpRequest](#) &request, [HttpResponse](#) &response, const QByteArray &key) const =0
Returns the property referenced by *key* in the session, or a null QVariant if the property isn't found.
- virtual void [setProperty](#) (const [HttpRequest](#) &request, [HttpResponse](#) &response, const QByteArray &key, const QVariant &value)=0
Sets the property's value referenced by *key* to *value*.
- virtual void [removeProperty](#) (const [HttpRequest](#) &request, [HttpResponse](#) &response, const QByteArray &key)=0
Removes the property referenced by *key* in the session, if any.
- void [setMacSecret](#) (const QByteArray &secret)
Sets the secret key of the message authentication code.

Static Public Member Functions

- static [SessionSettings](#) [defaultSettings](#) ()
Returns the default settings, used when you don't specify the settings argument in [SessionStore](#) constructor.

Protected Member Functions

- QByteArray [session](#) (const [HttpRequest](#) &request) const
Returns the value of the first cookie that is compatible with this store's properties, as defined in its settings.
- QByteArray [session](#) (const [HttpRequest](#) &request, const [HttpResponse](#) &response) const
Returns the value of the first cookie that is compatible with this store's properties, as defined in its settings.
- void [setSession](#) ([HttpResponse](#) &response, const QByteArray &[session](#)) const
Sets a cookie that matches the store's settings in the *response* object.
- void [unsetSession](#) ([HttpResponse](#) &response) const
Invalidates the user's session.

Protected Attributes

- [SessionSettings](#) [settings](#)
This attribute represents the session's settings.

10.24.1 Detailed Description

[SessionStore](#) class can be used to store data that must persist among different requests.

This is used in web applications, among other, to measure online users, create a shopping list per user and implement a login system.

Using sessions

The session data is stored in properties. You can create a session to any pair of request and response objects and each one will have an independent set of properties.

Note

You can configure a session in any response (including 400-level and 500-level status responses), but the user agent may ignore sessions with a 100-level status line.

To access the session properties, you must pass the request and response objects each time you want to manipulate, but this step can be eliminated using the [Session](#) helper class.

Note

You can configure multiple sessions to the same user agent, but only if you are using settings that have different scope. The settings have the same scope if the name, domain and path attributes are equal.

Warning

Despite the possibility of use different stores having SessionSetting objects with the same name attribute, it's recommended to avoid the use of this feature because a store don't have knowledge about the settings of other stores and can easily use the wrong request's cookie in a misconfigured application.

One session only persists for a specified period of time (set through [SessionSettings](#)) and the storage details depends upon the class implementing the [SessionStore](#) interface.

Note

The use of sessions don't preclude HTTP caches from storing and reusing a response.

Security concerns

Warning

Tufão uses cookies as base for its session support, but cookies have a number of security pitfalls and you should read this section if you are going to use them to store any sensitive information, such as login systems.

To better understand the problems shown in the following subsections, you should understand how cookies works.

A cookie is a piece of text built of a name and a value. The server includes a *Set-Cookie* header and the user agent will include this cookie in every subsequent request made to this server (when the cookie's scope applies), until the cookie expires.

For example, consider that user *U* made a request to server *S* and received the following reply:

```
HTTP/1.1 200 OK
Set-Cookie: Pants=On
...
```

Then, user *U* will include the following header in every request made to *S*:

```
Cookie: Pants=On
```

In [SessionStore](#), cookies are abstracted, and they'll be created as needed by some operations. If it finds a valid cookie, it'll use, but if not, it'll create one.

The "valid cookie" term depends on the implementation used.

Attacks using session fixation

These attacks exploits applications that allows one user fixate another's user cookie. Consider the following code:

```
bool RequestHandler::handleRequest(Tufao::HttpRequest &request,
                                  Tufao::HttpResponse &response)
{
    QByteArray username(getUsername(request));
    QByteArray userpassword(getUserpassword(request));

    if (!performLogin(username, userpassword)) {
        loginFail(response);
        return true;
    }

    store.setProperty(request, response, "user", username);

    loginSuccess(response);
    return true;
}
```

Try to answer: What could happen with an application running the previous code?

Here is what can happens:

1. The attacker fixate the victim cookie's value to "I_KNOW_YOUR_ID". Depending on the application store's settings (including implementation and mac secret), an arbitrary value will be rejected by the store, but the attacker may still be able get an usable cookie value (maybe using the application itself to generate one).
2. The victim access the application and perform the login.
3. The application's store finds a cookie and reuses it to set the *user* property.
4. The attacker uses the victim cookie's value and will have unlimited access to the victim's account.

How an attacker could set the victim cookie value is out of the scope of this document, because there are *several* techniques to achieve this.

Never trust users input.

How defend against session fixation

In Tufão, you can defend against session fixation calling [SessionStore::resetSession](#) to force a new session to be created.

```
bool RequestHandler::handleRequest(Tufao::HttpRequest &request,
                                  Tufao::HttpResponse &response)
{
    // To make sense, this line must be inserted before call any session-related
    // code
    store.resetSession(request);

    QByteArray username(getUsername(request));
    QByteArray userpassword(getUserpassword(request));

    if (!performLogin(username, userpassword)) {
        loginFail(response);
        return true;
    }

    store.setProperty(request, response, "user", username);

    loginSuccess(response);
    return true;
}
```


Attacks using cross site request forgery

After the user is authenticated in your application, it can perform some actions, such as transfer money to another account. If it's possible to create a script to automate these actions, then it's possible for an attacker to instruct the victim's user agent to perform this action. An example of a scriptable action is the url below:

```
http://bank.example.com/withdraw?account=foo&amount=1000000&for=bar
```

First, you *shouldn't* allow GET methods to mutate the server's state, but, in this case, use POST wouldn't defend our users against attackers. To prevent our actions from being scriptable, we need to require that an action only will be valid if it's originated from our application.

A technique to achieve the behaviour suggested in the previous paragraph is to generate some random data (a challenge token) and associate it with the user's current session. Then, we insert this token in the pages served to this user. The following code shows part of the solution:

```
bool RequestHandler::handleRequest(Tufao::HttpRequest &request,
                                   Tufao::HttpResponse &response)
{
    Tufao::Session s(store, request, response);

    // ...

    // Here, we provide needed data for the application to perform an action.
    // In the handler that will perform the action, we ignore requests with
    // incorrect tokens.
    response << /* ... */
               << "<input type=\"hidden\" name=\"CSRFToken\" value=\""
               << s["CSRFToken"] << "\">"
               << /* ... */;

    response.end();
    return true;
}
```

To generate a challenge token, you can use `QUuid`. To improve this design even further, you can regenerate the token and name parameters for each request.

Yet another option is to requiring the client to provide authentication data in the same request used to perform the action, but this solution add a usability issue.

Other problems

As defined in RFC 6265 and implemented in browsers, cookies don't provide strong confidentiality. They don't provide:

- Isolation by port. A service running on one port has access to cookies of a service running on another port of the same host.
- Isolation by scheme.
- Full-isolation by path. User agents won't send cookies from one path to another, but it's still possible for a service to *set* a cookie of a service running in another path of the same host.

Cookies don't provide strong integrity also.

You can avoid the previous problems by signing or encrypting cookies, but it'll still be possible for an attacker to perform a replay attack.

You can signing cookies in Tufão using [SessionStore::setMacSecret](#) and hide the session data using a [SessionStore](#) implementation that don't store its data in the cookie.

See [SessionSettings](#) for more information.

Implementing your own storage backend

If the implementations shipped with Tufão don't supply your needs, you can provide your own storage backend. To do that, you must implement the pure virtual methods of this class. They give you full access to the cookies returned from the user agent and included in the response objects.

The protected methods may help you process and include the cookies. They already take care of the boring task of read and comply with the [SessionSettings](#) object and sign the cookies.

This flexible design is what allows you, among other, implement a pure cookie based storage mechanism.

Security concerns

If the application request your store to manipulate some property and the request provides a session's id that don't correspond to any session in the store, you *should not* reuse this value as an id. You should *ignore it* and create a new one. This design may add restrictions to some forms of attacks.

See also

[SessionSettings](#) [Session](#)

Since

0.4

10.24.2 Constructor & Destructor Documentation

10.24.2.1 SessionStore()

```
Tufao::SessionStore::SessionStore (
    const SessionSettings & settings = defaultSettings(),
    QObject * parent = 0 ) [explicit]
```

Constructs a [SessionStore](#) object.

settings Specifies session parameters.

parent is passed to the QObject constructor.

10.24.3 Member Function Documentation

10.24.3.1 defaultSettings()

```
static SessionSettings Tufao::SessionStore::defaultSettings ( ) [static]
```

Returns the default settings, used when you don't specify the settings argument in [SessionStore](#) constructor.

The default values to these settings are:

- timeout: 15
- httpOnly: true
- key: "SID"
- path: "/"
- secure: false

If you use more than one store in the same application, you need to set a unique key to each one.

10.24.3.2 hasProperty()

```
virtual bool Tufao::SessionStore::hasProperty (
    const HttpRequest & request,
    const HttpResponse & response,
    const QByteArray & key ) const [pure virtual]
```

Returns true if the session has a property named key.

The session is represented by the pair request, response.

Implemented in [Tufao::SimpleSessionStore](#).

10.24.3.3 properties()

```
virtual QList<QByteArray> Tufao::SessionStore::properties (
    const HttpRequest & request,
    const HttpResponse & response ) const [pure virtual]
```

Returns a list of set properties to this session.

The session is represented by the pair request, response.

Implemented in [Tufao::SimpleSessionStore](#).

10.24.3.4 property()

```
virtual QVariant Tufao::SessionStore::property (
    const HttpRequest & request,
    HttpResponse & response,
    const QByteArray & key ) const [pure virtual]
```

Returns the property referenced by `key` in the session, or a null `QVariant` if the property isn't found.

The session is represented by the pair `request, response`.

Implemented in [Tufao::SimpleSessionStore](#).

10.24.3.5 removeProperty()

```
virtual void Tufao::SessionStore::removeProperty (
    const HttpRequest & request,
    HttpResponse & response,
    const QByteArray & key ) [pure virtual]
```

Removes the property referenced by `key` in the session, if any.

The session is represented by the pair `request, response`.

Implemented in [Tufao::SimpleSessionStore](#).

10.24.3.6 removeSession()

```
virtual void Tufao::SessionStore::removeSession (
    const HttpRequest & request,
    HttpResponse & response ) [pure virtual]
```

Removes the session, if any, in the `request`.

response The object used to invalidate the user's cookie.

Implemented in [Tufao::SimpleSessionStore](#).

10.24.3.7 resetSession()

```
void Tufao::SessionStore::resetSession (
    HttpRequest & request ) const
```

This method removes all cookies matching with this store's settings from `request`.

The purpose is hide the cookies from other pieces of code handling `request`.

Call this if you need to reset the session id.

10.24.3.8 session() [1/2]

```
QByteArray Tufao::SessionStore::session (
    const HttpRequest & request ) const [protected]
```

Returns the value of the first cookie that is compatible with this store's properties, as defined in its settings.

Warning

If you get a session that doesn't exist, you *MUST NOT* reuse this id. Create a new one and discard this, or you will introduce a session fixation vulnerability.

10.24.3.9 session() [2/2]

```
QByteArray Tufao::SessionStore::session (
    const HttpRequest & request,
    const HttpResponse & response ) const [protected]
```

Returns the value of the first cookie that is compatible with this store's properties, as defined in its settings.

Besides searching in the `request`'s headers named as "Cookie", search in `response`'s headers named as "Set-Cookie". This allows you to manipulate sessions as soon as they are set.

Warning

If you get a session that doesn't exist, you *MUST NOT* reuse this id. Create a new one and discard this, or you will introduce a session fixation vulnerability.

10.24.3.10 setMacSecret()

```
void Tufao::SessionStore::setMacSecret (
    const QByteArray & secret )
```

Sets the secret key of the message authentication code.

If set, it will protect the cookies integrity and authenticity.

Warning

The value used here *must* remain secret.

The algorithm used by Tufão is HMAC + SHA1.

10.24.3.11 `setProperty()`

```
virtual void Tufao::SessionStore::setProperty (
    const HttpRequest & request,
    HttpResponse & response,
    const QByteArray & key,
    const QVariant & value ) [pure virtual]
```

Sets the property's value referenced by `key` to `value`.

The session is represented by the pair `request`, `response`.

Implemented in [Tufao::SimpleSessionStore](#).

10.24.3.12 `setSession()`

```
void Tufao::SessionStore::setSession (
    HttpResponse & response,
    const QByteArray & session ) const [protected]
```

Sets a cookie that matches the store's settings in the `response` object.

Note

It will also renew cookie's lifetime.

Warning

Call this method more than once to the same `response` object has undefined behaviour.

Note

If you need to create a new unique identifier, but don't know how, check the `QUuid` class.

See also

[SessionSettings::cookie](#)

10.24.3.13 `unsetSession()`

```
void Tufao::SessionStore::unsetSession (
    HttpResponse & response ) const [protected]
```

Invalidates the user's session.

This is done by setting the expiration date to the past, as specified by RFC 6265.

10.24.4 Member Data Documentation

10.24.4.1 settings

`SessionSettings` Tufao::SessionStore::settings [protected]

This attribute represents the session's settings.

It will be used in the operations involving cookie's handling and is set automatically in [SessionStore](#)'s constructor.

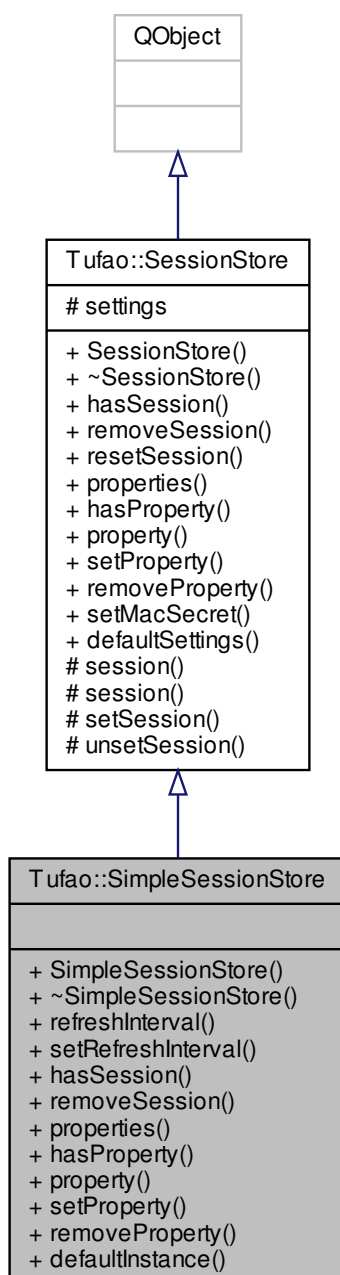
The documentation for this class was generated from the following file:

- sessionstore.h

10.25 Tufao::SimpleSessionStore Class Reference

[SimpleSessionStore](#) implements a simple storage mechanism to [SessionStore](#).

Inheritance diagram for Tufao::SimpleSessionStore:



Collaboration diagram for Tufao::SimpleSessionStore:



Public Member Functions

- [SimpleSessionStore](#) (const [SessionSettings](#) &settings=defaultSettings(), QObject *parent=0)
Constructs a new [SimpleSessionStore](#) object.
- [~SimpleSessionStore](#) ()
Destructs a [SimpleSessionStore](#) object.
- int [refreshInterval](#) () const

The refresh interval used to look for (and delete) expired cookies.

- void [setRefreshInterval](#) (int msec)
- Sets the refresh interval used to look for (and delete) expired cookies.*
- bool [hasSession](#) (const [HttpRequest](#) &request) const override
- Implements [SessionStore::hasSession](#).*
- void [removeSession](#) (const [HttpRequest](#) &request, [HttpResponse](#) &response) override
- Implements [SessionStore::removeSession](#).*
- QList< QByteArray > [properties](#) (const [HttpRequest](#) &request, const [HttpResponse](#) &response) const override
- Implements [SessionStore::properties](#).*
- bool [hasProperty](#) (const [HttpRequest](#) &request, const [HttpResponse](#) &response, const QByteArray &key) const override
- Implements [SessionStore::hasProperty](#).*
- QVariant [property](#) (const [HttpRequest](#) &request, [HttpResponse](#) &response, const QByteArray &key) const override
- Implements [SessionStore::property](#).*
- void [setProperty](#) (const [HttpRequest](#) &request, [HttpResponse](#) &response, const QByteArray &key, const QVariant &value) override
- Implements [SessionStore::setProperty](#).*
- void [removeProperty](#) (const [HttpRequest](#) &request, [HttpResponse](#) &response, const QByteArray &key) override
- Implements [SessionStore::removeProperty](#).*

Static Public Member Functions

- static [SimpleSessionStore](#) & [defaultInstance](#) ()
- Returns a reference to the same store every time it's called.*

Additional Inherited Members

10.25.1 Detailed Description

[SimpleSessionStore](#) implements a simple storage mechanism to [SessionStore](#).

It will store the data in the system memory.

It will look for expired cookies (and delete them) at a defined interval (the default value is `DEFAULT_REFRESH_INTERVAL`).

Since

0.4

10.25.2 Constructor & Destructor Documentation

10.25.2.1 SimpleSessionStore()

```
Tufao::SimpleSessionStore::SimpleSessionStore (
    const SessionSettings & settings = defaultSettings(),
    QObject * parent = 0 ) [explicit]
```

Constructs a new [SimpleSessionStore](#) object.

It will pass `parent` to `QObject` constructor and `settings` to [SessionStore](#) constructor.

10.25.3 Member Function Documentation

10.25.3.1 defaultInstance()

```
static SimpleSessionStore& Tufao::SimpleSessionStore::defaultInstance ( ) [static]
```

Returns a reference to the same store every time it's called.

It acts like a [singleton](#).

Note

This method is added for convenience and if you are developing multithreaded applications you shouldn't use it. But, if you insist use it for these applications, you can use a mutex to control its access.

10.25.3.2 refreshInterval()

```
int Tufao::SimpleSessionStore::refreshInterval ( ) const
```

The refresh interval used to look for (and delete) expired cookies.

The default interval is the value of the macro `DEFAULT_REFRESH_INTERVAL`.

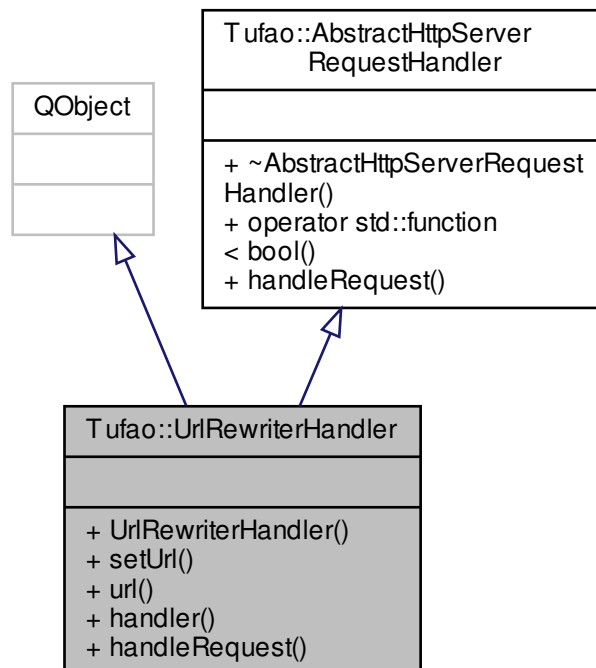
The documentation for this class was generated from the following file:

- `simplesessionstore.h`

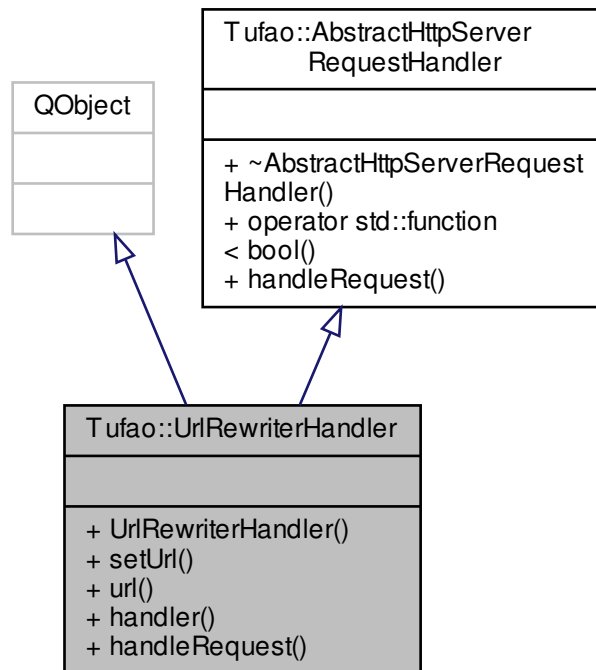
10.26 Tufao::UrlRewriterHandler Class Reference

This class provides a handler to internally (only seen by your application) rewrite the URL.

Inheritance diagram for Tufao::UrlRewriterHandler:



Collaboration diagram for Tufao::UrlRewriterHandler:



Public Slots

- bool [handleRequest](#) (Tufao::HttpServerRequest &request, Tufao::HttpServerResponse &response) override
Changes the `request` url and returns false.

Public Member Functions

- [UrlRewriterHandler](#) (const QUrl &url, QObject *parent=0)
Constructs a [UrlRewriterHandler](#) object.
- void [setUrl](#) (const QUrl &url)
Sets the URL that will replace the old URL to `url`.
- QUrl [url](#) () const
Returns the url used to replace the old URLs.

Static Public Member Functions

- static std::function< bool([HttpServerRequest](#) &, [HttpServerResponse](#) &)> [handler](#) (const QUrl &url)
Returns a handler that don't depends on another object.

10.26.1 Detailed Description

This class provides a handler to internally (only seen by your application) rewrite the URL.

Note

The handler does **NOT** redirects the HTTP client to another path. The new URL can only be seen by the Tufão application itself and HTTP clients will think they are receiving a response to the path they originally asked for.

One place where this technique is useful is when you want to use one file to handle the root page of your site through [HttpFileServer](#). This example is illustrated in the image below:

Since

0.6

10.26.2 Constructor & Destructor Documentation

10.26.2.1 UrlRewriterHandler()

```
Tufao::UrlRewriterHandler::UrlRewriterHandler (
    const QUrl & url,
    QObject * parent = 0 ) [explicit]
```

Constructs a [UrlRewriterHandler](#) object.

`parent` is passed to the `QObject` constructor.

`url` will be the new url.

Since

1.0

10.26.3 Member Function Documentation

10.26.3.1 handler()

```
static std::function<bool(HttpServerRequest&, HttpServerResponse&)> Tufao::UrlRewriterHandler↔
::handler (
    const QUrl & url ) [static]
```

Returns a handler that don't depends on another object.

The purpose of this alternative handler is to free you of the worry of maintain the [UrlRewriterHandler](#)'s object (lifetime) while the functor object is being used.

Since

1.0

10.26.3.2 `handleRequest`

```
bool Tufao::UrlRewriterHandler::handleRequest (
    Tufao::HttpRequest & request,
    Tufao::HttpResponse & response ) [override], [slot]
```

Changes the `request` url and returns false.

This method doesn't do anything else.

Since

1.0

10.26.3.3 `setUrl()`

```
void Tufao::UrlRewriterHandler::setUrl (
    const QUrl & url )
```

Sets the URL that will replace the old URL to `url`.

Since

1.0

10.26.3.4 `url()`

```
QUrl Tufao::UrlRewriterHandler::url ( ) const
```

Returns the url used to replace the old URLs.

Since

1.0

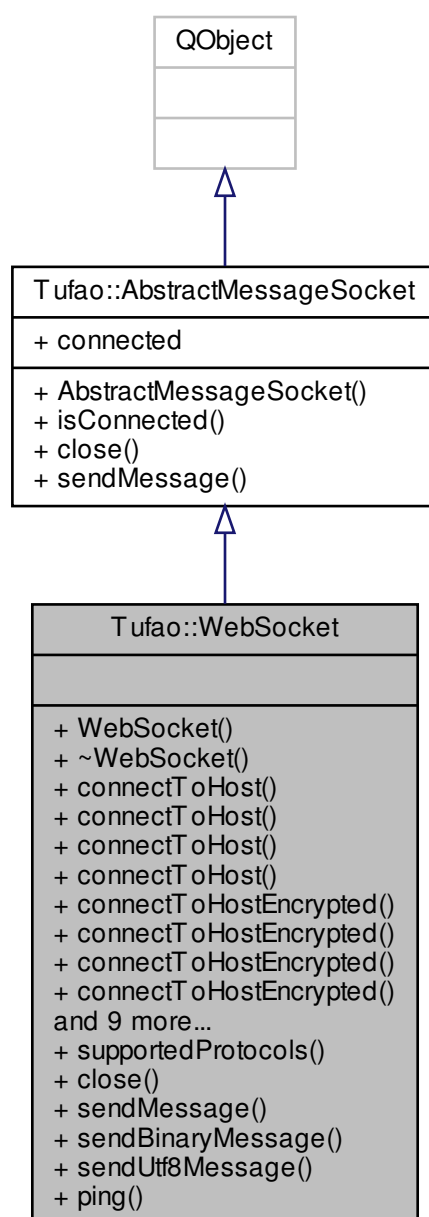
The documentation for this class was generated from the following file:

- `urlrewriterhandler.h`

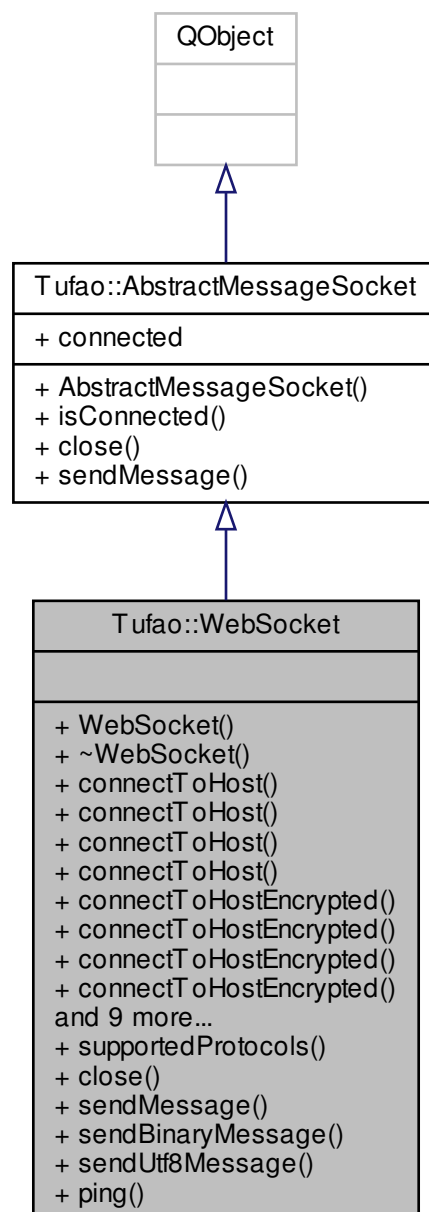
10.27 Tufao::WebSocket Class Reference

This class represents a [WebSocket](#) connection.

Inheritance diagram for Tufao::WebSocket:



Collaboration diagram for Tufao::WebSocket:



Public Slots

- void **close** () override
- bool **sendMessage** (const QByteArray &msg) override
- bool **sendBinaryMessage** (const QByteArray &msg)
Sends a binary message over the connection.
- bool **sendUtf8Message** (const QByteArray &msg)
Sends a UTF-8 text message over the connection.

- bool [ping](#) (const QByteArray &data)
Sends a ping frame over the connection.

Signals

- void [pong](#) (QByteArray data)
This signal is emitted when a pong frame is received.

Public Member Functions

- [WebSocket](#) (QObject *parent=0)
Constructs a [Tufao::WebSocket](#) object.
- [~WebSocket](#) ()
Destroys the object.
- bool [connectToHost](#) (const QHostAddress &address, quint16 port, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
*Execute the steps to establish a [WebSocket](#) connection with the server represented by *address* and *port* on the given *resource*.*
- bool [connectToHost](#) (const QHostAddress &address, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- bool [connectToHost](#) (const QString &hostname, quint16 port, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- bool [connectToHost](#) (const QString &hostname, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- bool [connectToHostEncrypted](#) (const QString &hostname, quint16 port, const QByteArray &resource, const [Headers](#) &headers, const QList< QSslError > &ignoredSslErrors)
The same as [WebSocket::connectToHost](#), but uses a TLS connection.
- bool [connectToHostEncrypted](#) (const QString &hostname, quint16 port, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- bool [connectToHostEncrypted](#) (const QString &hostname, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- bool [connectToHostEncrypted](#) (const QHostAddress &address, quint16 port, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- bool [connectToHostEncrypted](#) (const QHostAddress &address, const QByteArray &resource, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- bool [startServerHandshake](#) (const [HttpRequest](#) &request, const QByteArray &head, const [Headers](#) &headers=[Headers](#)())
*It establish a [WebSocket](#) connection initiated by *request* with *head* data.*
- bool [startServerHandshake](#) ([HttpRequest](#) &request, const [Headers](#) &headers=[Headers](#)())
This is an overloaded function.
- void [setMessageType](#) ([WebSocketMessageType](#) type)
Set the type of messages sent through [WebSocket::sendMessage](#) method.
- [WebSocketMessageType](#) [messageType](#) () const
Return the current type of messages what will be sent through [WebSocket::sendMessage](#) method.
- [WebSocketError](#) [error](#) () const

- Returns the type of last error that occurred.*
- QString [errorString](#) () const
Returns a human-readable description of the last error that occurred.
- QHostAddress [peerAddress](#) () const
Returns the address of the connected peer.
- quint16 [peerPort](#) () const
Returns the port of the connected peer.

Static Public Member Functions

- static QList< QByteArray > [supportedProtocols](#) (const [Headers](#) &headers)
It reads the supported sub-protocols from the appropriate fields and return them.

Additional Inherited Members

10.27.1 Detailed Description

This class represents a [WebSocket](#) connection.

[WebSocket](#) is a protocol designed to allow HTTP user agents and servers communicates using a two-way protocol. It's possible to upgrade an established HTTP connection to a [WebSocket](#) connection.

[WebSocket](#) server

If you intend to create a server able to accept [WebSocket](#) connections, you must create a HTTP server and create a handler to upgrade events.

In the handler for the upgrade event, you must create a new [WebSocket](#) object and call the method `startServerHandshake`. This method will send the initial [WebSocket](#) server payload and check if it's a valid [WebSocket](#) connection request.

[WebSocket](#) client

If you intend to connect to a [WebSocket](#) server, you must call one of the `connectToHost` methods. If the connection should be encrypted, then call one of the `connectToHostEncrypted` methods. The `connected` signal will emitted when the socket is ready.

See also

[Tufao::AbstractMessageSocket](#)

Since

0.2

10.27.2 Constructor & Destructor Documentation

10.27.2.1 WebSocket()

```
Tufao::WebSocket::WebSocket (
    QObject * parent = 0 ) [explicit]
```

Constructs a [Tufao::WebSocket](#) object.

`parent` is passed to the `QObject` constructor.

10.27.3 Member Function Documentation

10.27.3.1 connectToHost() [1/4]

```
bool Tufao::WebSocket::connectToHost (
    const QHostAddress & address,
    quint16 port,
    const QByteArray & resource,
    const Headers & headers = Headers() )
```

Execute the steps to establish a [WebSocket](#) connection with the server represented by `address` and `port` on the given `resource`.

You can send extra headers in the [WebSocket](#) opening handshake request using the `headers` argument.

Note

Optional headers such as `Origin` aren't added by default. If you want to use one of the headers, you should add them explicitly. The list of common optional headers is the following:

- `Origin`: RFC 6454. If your software is a browser client, this field is mandatory, not optional.
- `Sec-WebSocket-Protocol`: If present, this value indicates one or more comma-separated subprotocol the client wishes to speak, ordered by preference.

If the object fail to establish a connection, it will emit `disconnected` signal. If it proceeds, it will emit the `connected` signal.

10.27.3.2 connectToHost() [2/4]

```
bool Tufao::WebSocket::connectToHost (
    const QHostAddress & address,
    const QByteArray & resource,
    const Headers & headers = Headers() )
```

This is an overloaded function.

It uses port 80 to establish the connection.

10.27.3.3 connectToHost() [3/4]

```
bool Tufao::WebSocket::connectToHost (
    const QString & hostname,
    quint16 port,
    const QByteArray & resource,
    const Headers & headers = Headers() )
```

This is an overloaded function.

Since

0.3

10.27.3.4 connectToHost() [4/4]

```
bool Tufao::WebSocket::connectToHost (
    const QString & hostname,
    const QByteArray & resource,
    const Headers & headers = Headers() )
```

This is an overloaded function.

It uses port 80 to establish the connection.

Since

0.3

10.27.3.5 connectToHostEncrypted() [1/4]

```
bool Tufao::WebSocket::connectToHostEncrypted (
    const QString & hostname,
    quint16 port,
    const QByteArray & resource,
    const Headers & headers,
    const QList< QSSLError > & ignoredSslErrors )
```

The same as [WebSocket::connectToHost](#), but uses a TLS connection.

`ignoredSslErrors` is passed to `QSslSocket::ignoreSslErrors`.

See also

[Tufao::WebSocket::connectToHost](#)

Since

1.1

10.27.3.6 `connectToHostEncrypted()` [2/4]

```
bool Tufao::WebSocket::connectToHostEncrypted (
    const QString & hostname,
    const QByteArray & resource,
    const Headers & headers = Headers() )
```

This is an overloaded function.

It uses port 443 to establish the connection.

10.27.3.7 `connectToHostEncrypted()` [3/4]

```
bool Tufao::WebSocket::connectToHostEncrypted (
    const QHostAddress & address,
    quint16 port,
    const QByteArray & resource,
    const Headers & headers = Headers() )
```

This is an overloaded function.

See also

[Tufao::WebSocket::connectToHost](#)

Since

0.3

10.27.3.8 `connectToHostEncrypted()` [4/4]

```
bool Tufao::WebSocket::connectToHostEncrypted (
    const QHostAddress & address,
    const QByteArray & resource,
    const Headers & headers = Headers() )
```

This is an overloaded function.

It uses port 443 to establish the connection.

Since

0.3

10.27.3.9 messagesType()

```
WebSocketMessageType Tufao::WebSocket::messagesType ( ) const
```

Return the current type of messages what will be sent through `WebSocket::sendMessage` method.

See also

[Tufao::WebSocket::setMessagesType](#)

10.27.3.10 peerAddress()

```
QHostAddress Tufao::WebSocket::peerAddress ( ) const
```

Returns the address of the connected peer.

Return values

<code>QHostAddress::Null</code>	if the socket is not in <code>ConnectedState</code> .
---------------------------------	--

Since

0.5

10.27.3.11 `peerPort()`

```
quint16 Tufao::WebSocket::peerPort ( ) const
```

Returns the port of the connected peer.

Return values

<code>0</code>	if the socket is not in <code>ConnectedState</code> .
----------------	--

Since

0.5

10.27.3.12 `ping`

```
bool Tufao::WebSocket::ping (
    const QByteArray & data ) [slot]
```

Sends a ping frame over the connection.

A [WebSocket](#) peer that receives a ping frame must respond with a pong frame as soon as practical.

You can use a ping frame to measure the connection lag, to test the connectivity availability, among other.

10.27.3.13 `pong`

```
void Tufao::WebSocket::pong (
    QByteArray data ) [signal]
```

This signal is emitted when a pong frame is received.

Pong frames are sent in response to ping frames.

Note

This signal is unsafe (read this: [Safe signals](#))!

See also

[WebSocket::ping](#)

10.27.3.14 sendBinaryMessage

```
bool Tufao::WebSocket::sendBinaryMessage (
    const QByteArray & msg ) [slot]
```

Sends a binary message over the connection.

See also

WebSocket::sendMessage [WebSocket::messagesType](#)

10.27.3.15 sendUtf8Message

```
bool Tufao::WebSocket::sendUtf8Message (
    const QByteArray & msg ) [slot]
```

Sends a UTF-8 text message over the connection.

See also

WebSocket::sendMessage [WebSocket::messagesType](#)

10.27.3.16 setMessagesType()

```
void Tufao::WebSocket::setMessagesType (
    WebSocketMessageType type )
```

Set the type of messages sent through `WebSocket::sendMessage` method.

Note

Another way of choose the type of sent messages is through [WebSocket::sendBinaryMessage](#) and [WebSocket::sendUtf8Message](#) methods.
The default value is `BINARY_MESSAGE`

10.27.3.17 startServerHandshake() [1/2]

```
bool Tufao::WebSocket::startServerHandshake (
    const HttpRequest & request,
    const QByteArray & head,
    const Headers & headers = Headers() )
```

It establish a [WebSocket](#) connection initiated by `request` with `head` data.

The method send the initial [WebSocket](#) server payload and check if it's a valid [WebSocket](#) connection request.

Note

You should call this function only after `request` emits the [Tufao::HttpRequest::upgrade](#) signal.

Note

[Tufao::WebSocket](#) won't treat optional headers found in `request`. If you want to respond to these headers in the opening handshake response, just put the headers you want to send in the response in the `headers` argument. Some headers you may want to treat are:

- Origin: Compare the Origin to the Host to block malicious scripts coming from web browsers.
- Cookie
- Sec-WebSocket-Protocol: The subprotocol the client wishes to speak.

Note

If you want to perform additional client authentication, you should start the handshake only after the authentication occurs. You can use the 401 status code with "WWW-Authenticate" header to perform the authentication, among other methods.

Note

You can use a 3xx status code to redirect the client.

Note

If the handshake fail, the method will write the appropriate message to the socket and return false.

Since

1.0

10.27.3.18 startServerHandshake() [2/2]

```
bool Tufao::WebSocket::startServerHandshake (
    HttpRequest & request,
    const Headers & headers = Headers() )
```

This is an overloaded function.

It uses [HttpRequest::readBody](#) to figure out the *head* argument.

Since

1.0

The documentation for this class was generated from the following file:

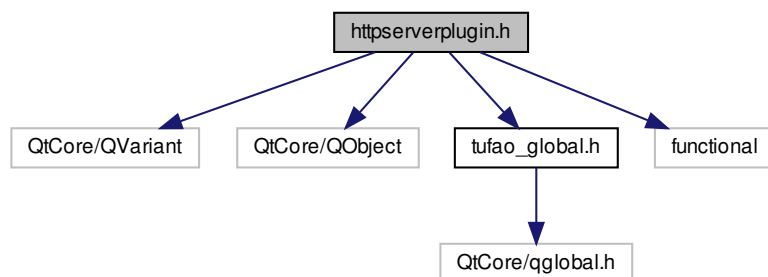
- websocket.h

Chapter 11

File Documentation

11.1 httpserverplugin.h File Reference

Include dependency graph for httpserverplugin.h:



Classes

- class [Tufao::HttpServerPlugin](#)

This class provides a factory interface to create request handlers and communicate with factories of other plugins.

Namespaces

- [Tufao](#)

This is the namespace where all Tufão facilities are grouped.

Macros

- `#define` [TUFAO_HTTPSERVERPLUGIN_IID](#) "Tufao::HttpServerPlugin/1.0"

This definition stores the [Tufao::HttpServerPlugin](#) interface IID.

11.1.1 Macro Definition Documentation

11.1.1.1 TUFAO_HTTPSERVERPLUGIN_IID

```
#define TUFAO_HTTPSERVERPLUGIN_IID "Tufao::HttpServerPlugin/1.0"
```

This definition stores the [Tufao::HttpServerPlugin](#) interface IID.

When you define a plugin, you'll use this macro as the IID argument of `Q_PLUGIN_METADATA` macro to tell the class is implementing the [Tufao::HttpServerPlugin](#) interface. Example:

```
#ifndef PLUGIN_H
#define PLUGIN_H

#include <Tufao/HttpServerPlugin>

class Plugin: public QObject, Tufao::HttpServerPlugin
{
    Q_OBJECT
    Q_PLUGIN_METADATA(IID TUFAO_HTTPSERVERPLUGIN_IID)
    Q_INTERFACES(Tufao::HttpServerPlugin)
public:
    std::function<bool(Tufao::HttpServerRequest&, Tufao::HttpServerResponse&)>
    createHandler(const QHash<QString, Tufao::HttpServerPlugin*> &dependencies,
                 const QVariant &customData = QVariant()) override;
};

#endif // PLUGIN_H
```

Index

- AbstractMessageSocket
 - Tufao::AbstractMessageSocket, 33
- addPluginLocation
 - Tufao::ClassHandlerManager, 41
- addTrailer
 - Tufao::HttpServerResponse, 88
- addTrailers
 - Tufao::HttpServerResponse, 88
- apply
 - Tufao::Session, 112, 113
- bufferSize
 - Tufao::HttpFileServer, 48
- canHandleRequest
 - Tufao::HttpFileServer, 48
- checkContinue
 - Tufao::HttpServer, 62
- ClassHandlerManager
 - Tufao::ClassHandlerManager, 40
- close
 - Tufao::HttpServer, 62
 - Tufao::HttpRequest, 70
- config
 - Tufao::HttpPluginServer, 54
- connectToHost
 - Tufao::WebSocket, 144, 145
- connectToHostEncrypted
 - Tufao::WebSocket, 145, 146
- connected
 - Tufao::AbstractMessageSocket, 33
- createHandler
 - Tufao::HttpServerPlugin, 66
- customData
 - Tufao::HttpRequest, 70
- data
 - Tufao::HttpRequest, 70
- defaultInstance
 - Tufao::SimpleSessionStore, 135
- defaultSettings
 - Tufao::SessionStore, 126
- defaultUpgradeHandler
 - Tufao::HttpServer, 62
- disconnected
 - Tufao::AbstractMessageSocket, 33
- domain
 - Tufao::SessionSettings, 116
- end
 - Tufao::HttpRequest, 71
 - Tufao::HttpServerResponse, 89
- finished
 - Tufao::HttpServerResponse, 89
- flush
 - Tufao::HttpServerResponse, 90
- fromDateTime
 - Tufao::Headers, 43
- handleConnection
 - Tufao::HttpServer, 62
- handleRequest
 - Tufao::AbstractHttpRequestHandler, 26
 - Tufao::HttpFileServer, 49
 - Tufao::HttpPluginServer, 54
 - Tufao::HttpRequestRouter, 81
 - Tufao::UrlRewriterHandler, 138
- handleUpgrade
 - Tufao::AbstractHttpUpgradeHandler, 29
 - Tufao::HttpUpgradeRouter, 100
- Handler
 - Tufao::HttpRequestRouter, 81
 - Tufao::HttpUpgradeRouter, 100
- handler
 - Tufao::HttpFileServer, 49
 - Tufao::NotFoundHandler, 108
 - Tufao::UrlRewriterHandler, 138
- hasProperty
 - Tufao::SessionStore, 127
- hasValue
 - Tufao::Session, 113
- headers
 - Tufao::HttpRequest, 71
 - Tufao::HttpServerResponse, 90
- HttpFileServer
 - Tufao::HttpFileServer, 48
- httpOnly
 - Tufao::SessionSettings, 117
- HttpPluginServer
 - Tufao::HttpPluginServer, 53
- HttpResponseStatus
 - Tufao, 21
- HttpServer
 - Tufao::HttpServer, 61
- HttpRequest
 - Tufao::HttpRequest, 70
- HttpRequestRouter
 - Tufao::HttpRequestRouter, 81
- HttpServerResponse

- Tufao::HttpServerResponse, 87
- httpserverplugin.h, 153
 - TUFAO_HTTPSERVERPLUGIN_IID, 154
- incomingConnection
 - Tufao::HttpServer, 63
- listen
 - Tufao::HttpServer, 63
- map
 - Tufao::HttpServerRequestRouter, 82, 83
 - Tufao::HttpUpgradeRouter, 101
- messagesType
 - Tufao::WebSocket, 146
- method
 - Tufao::HttpRequest, 72
 - Tufao::HttpServerRequestRouter::Mapping, 106
- name
 - Tufao::SessionSettings, 117
- newMessage
 - Tufao::AbstractMessageSocket, 34
- NotFoundHandler
 - Tufao::NotFoundHandler, 108
- operator std::function< bool
 - Tufao::AbstractHttpRequestHandler, 27
 - Tufao::AbstractHttpUpgradeHandler, 30
- operator std::function< void
 - Tufao::AbstractHttpUpgradeHandler, 30
- operator<<
 - Tufao, 23
- Option
 - Tufao::HttpServerResponse, 87
- path
 - Tufao::HttpServerRequestRouter::Mapping, 106
 - Tufao::HttpUpgradeRouter::Mapping, 104
 - Tufao::SessionSettings, 117
- peerAddress
 - Tufao::WebSocket, 147
- peerPort
 - Tufao::WebSocket, 148
- ping
 - Tufao::WebSocket, 148
- pong
 - Tufao::WebSocket, 148
- properties
 - Tufao::SessionStore, 127
- property
 - Tufao::SessionStore, 127
- PropertyWrapper
 - Tufao::Session::PropertyWrapper, 110
- readBody
 - Tufao::HttpRequest, 73
- ready
 - Tufao::HttpRequest, 73
- refreshInterval
 - Tufao::SimpleSessionStore, 135
- removeProperty
 - Tufao::SessionStore, 128
- removeSession
 - Tufao::SessionStore, 128
- requestReady
 - Tufao::HttpServer, 63
- resetSession
 - Tufao::SessionStore, 128
- responseOptions
 - Tufao::HttpRequest, 74
- resume
 - Tufao::HttpRequest, 74
- secure
 - Tufao::SessionSettings, 118
- sendBinaryMessage
 - Tufao::WebSocket, 148
- sendMessage
 - Tufao::AbstractMessageSocket, 34
- sendUtf8Message
 - Tufao::WebSocket, 149
- serveFile
 - Tufao::HttpFileServer, 49, 50
- serverPort
 - Tufao::HttpServer, 64
- Session
 - Tufao::Session, 112
- session
 - Tufao::SessionStore, 128, 129
- SessionStore
 - Tufao::SessionStore, 126
- setConfig
 - Tufao::HttpPluginServer, 54
- setCustomData
 - Tufao::HttpRequest, 75
- setDir
 - Tufao::HttpFileServer, 50
- setLocalCertificate
 - Tufao::HttpsServer, 96
- setMacSecret
 - Tufao::SessionStore, 129
- setMessageType
 - Tufao::WebSocket, 149
- setOptions
 - Tufao::HttpServerResponse, 90
- setPrivateKey
 - Tufao::HttpsServer, 96
- setProperty
 - Tufao::SessionStore, 129
- setSession
 - Tufao::SessionStore, 130
- setTimeout
 - Tufao::HttpServer, 64
 - Tufao::HttpRequest, 75
- setUpgradeHandler
 - Tufao::HttpServer, 64
- setUrl
 - Tufao::HttpRequest, 75

- Tufao::UrlRewriterHandler, 139
- setValue
 - Tufao::Session, 114
- settings
 - Tufao::SessionStore, 131
- SimpleSessionStore
 - Tufao::SimpleSessionStore, 134
- socket
 - Tufao::HttpRequest, 76
- startServerHandshake
 - Tufao::WebSocket, 149, 150
- TUFAO_HTTPSERVERPLUGIN_IID
 - httpserverplugin.h, 154
- timeout
 - Tufao::SessionSettings, 118
- toDateTime
 - Tufao::Headers, 44
- trailers
 - Tufao::HttpRequest, 76
- Tufao, 19
 - HttpResponseStatus, 21
 - operator<<, 23
 - WebSocketError, 21
 - WebSocketMessageType, 23
- Tufao::AbstractHttpRequestHandler, 25
 - handleRequest, 26
 - operator std::function< bool, 27
- Tufao::AbstractHttpUpgradeHandler, 27
 - handleUpgrade, 29
 - operator std::function< bool, 30
 - operator std::function< void, 30
- Tufao::AbstractMessageSocket, 31
 - AbstractMessageSocket, 33
 - connected, 33
 - disconnected, 33
 - newMessage, 34
 - sendMessage, 34
- Tufao::ClassHandler, 35
- Tufao::ClassHandlerManager, 38
 - addPluginLocation, 41
 - ClassHandlerManager, 40
- Tufao::ClassHandlerPluginInfo, 41
- Tufao::Headers, 42
 - fromDateTime, 43
 - toDateTime, 44
- Tufao::HttpFileServer, 45
 - bufferSize, 48
 - canHandleRequest, 48
 - handleRequest, 49
 - handler, 49
 - HttpFileServer, 48
 - serveFile, 49, 50
 - setDir, 50
- Tufao::HttpPluginServer, 51
 - config, 54
 - handleRequest, 54
 - HttpPluginServer, 53
 - setConfig, 54
- Tufao::HttpServer, 57
 - checkContinue, 62
 - close, 62
 - defaultUpgradeHandler, 62
 - handleConnection, 62
 - HttpServer, 61
 - incomingConnection, 63
 - listen, 63
 - requestReady, 63
 - serverPort, 64
 - setTimeout, 64
 - setUpgradeHandler, 64
 - UpgradeHandler, 61
- Tufao::HttpServerPlugin, 65
 - createHandler, 66
- Tufao::HttpRequest, 67
 - close, 70
 - customData, 70
 - data, 70
 - end, 71
 - headers, 71
 - HttpRequest, 70
 - method, 72
 - readBody, 73
 - ready, 73
 - responseOptions, 74
 - resume, 74
 - setCustomData, 75
 - setTimeout, 75
 - setUrl, 75
 - socket, 76
 - trailers, 76
 - upgrade, 76
 - url, 77
- Tufao::HttpRequestRouter, 78
 - handleRequest, 81
 - Handler, 81
 - HttpRequestRouter, 81
 - map, 82, 83
 - unmap, 83
- Tufao::HttpRequestRouter::Mapping, 105
 - method, 106
 - path, 106
- Tufao::HttpResponse, 84
 - addTrailer, 88
 - addTrailers, 88
 - end, 89
 - finished, 89
 - flush, 90
 - headers, 90
 - HttpResponse, 87
 - Option, 87
 - setOptions, 90
 - write, 91
 - writeContinue, 91
 - writeHead, 92, 93
- Tufao::HttpUpgradeRouter, 97
 - handleUpgrade, 100

- Handler, 100
- map, 101
- Tufao::HttpUpgradeRouter::Mapping, 103
- path, 104
- Tufao::HttpsServer, 94
 - setLocalCertificate, 96
 - setPrivateKey, 96
- Tufao::IByteArray, 102
- Tufao::NotFoundHandler, 106
 - handler, 108
 - NotFoundHandler, 108
- Tufao::Session, 110
 - apply, 112, 113
 - hasValue, 113
 - Session, 112
 - setValue, 114
 - value, 114
- Tufao::Session::PropertyWrapper, 109
 - PropertyWrapper, 110
- Tufao::SessionSettings, 115
 - domain, 116
 - httpOnly, 117
 - name, 117
 - path, 117
 - secure, 118
 - timeout, 118
- Tufao::SessionStore, 119
 - defaultSettings, 126
 - hasProperty, 127
 - properties, 127
 - property, 127
 - removeProperty, 128
 - removeSession, 128
 - resetSession, 128
 - session, 128, 129
 - SessionStore, 126
 - setMacSecret, 129
 - setProperty, 129
 - setSession, 130
 - settings, 131
 - unsetSession, 130
- Tufao::SimpleSessionStore, 131
 - defaultInstance, 135
 - refreshInterval, 135
 - SimpleSessionStore, 134
- Tufao::UrlRewriterHandler, 136
 - handleRequest, 138
 - handler, 138
 - setUrl, 139
 - url, 139
 - UrlRewriterHandler, 138
- Tufao::WebSocket, 140
 - connectToHost, 144, 145
 - connectToHostEncrypted, 145, 146
 - messagesType, 146
 - peerAddress, 147
 - peerPort, 148
 - ping, 148
 - pong, 148
 - sendBinaryMessage, 148
 - sendUtf8Message, 149
 - setMessagesType, 149
 - startServerHandshake, 149, 150
 - WebSocket, 143
- unmap
 - Tufao::HttpServerRequestRouter, 83
- unsetSession
 - Tufao::SessionStore, 130
- upgrade
 - Tufao::HttpServerRequest, 76
- UpgradeHandler
 - Tufao::HttpServer, 61
- url
 - Tufao::HttpServerRequest, 77
 - Tufao::UrlRewriterHandler, 139
- UrlRewriterHandler
 - Tufao::UrlRewriterHandler, 138
- value
 - Tufao::Session, 114
- WebSocket
 - Tufao::WebSocket, 143
- WebSocketError
 - Tufao, 21
- WebSocketMessageType
 - Tufao, 23
- write
 - Tufao::HttpServerResponse, 91
- writeContinue
 - Tufao::HttpServerResponse, 91
- writeHead
 - Tufao::HttpServerResponse, 92, 93