<http://docs.mitk.org/nightly/RESTModule.html>

REST Module

**Table of Contents**

* [Description](http://docs.mitk.org/nightly/RESTModule.html#REST_brief)
	+ [Technical background](http://docs.mitk.org/nightly/RESTModule.html#REST_Technical)
* [How to use the REST Module](http://docs.mitk.org/nightly/RESTModule.html#Use_REST)
	+ [Use from a Server perspective](http://docs.mitk.org/nightly/RESTModule.html#Server_Use)
	+ [Use from a Client perspective](http://docs.mitk.org/nightly/RESTModule.html#Client_Use)

**Description**

The MITK REST Module is able to manage REST requests. The main class is the RESTManager. It is a MicroServices, which can be accessed via

auto \*context = [us::GetModuleContext](http://docs.mitk.org/nightly/group__MicroServices.html#ga68d5120d16225ab57e5edd9f19f2fec4)();

auto managerRef = context->GetServiceReference<IRESTManager>();

if (managerRef)

{

auto managerService = context->GetService(managerRef);

if (managerService)

{

//call the function you need from the service

}

}

**Technical background**

The module uses the [Microsoft C++ REST SDK](https://github.com/Microsoft/cpprestsdk) for REST mechanisms, as well as JSON conversion and asynchronous programming.

**How to use the REST Module**

You can use the REST module from two different perspectives in MITK:

1. The Server view (receive requests from clients)
2. The Client view (send requests to servers)

The following sections will give you an introduction on how to use which of those roles.

**Use from a Server perspective**

To act as a server, you need to implement the IRESTObserver, which has a Notify() method that has to be implemented. In this Notify() method you specify how you want to react to incoming requests and with which data you want to respond to the requests.

You can then start listening for requests from clients as shown below:

auto \*context = [us::GetModuleContext](http://docs.mitk.org/nightly/group__MicroServices.html#ga68d5120d16225ab57e5edd9f19f2fec4)();

auto managerRef = context->GetServiceReference<IRESTManager>();

if (managerRef)

{

auto managerService = context->GetService(managerRef);

if (managerService)

{

managerService->ReceiveRequests(uri /\*specify your uri which you want to receive requests for\*/, this);

}

}

If a client sends a request, the Notify method is called and a response is sent. By now, only GET-requests from clients are supported.

If you want to stop listening for requests, you can do this by calling

auto \*context = [us::GetModuleContext](http://docs.mitk.org/nightly/group__MicroServices.html#ga68d5120d16225ab57e5edd9f19f2fec4)();

auto managerRef = context->GetServiceReference<IRESTManager>();

if (managerRef)

{

auto managerService = context->GetService(managerRef);

if (managerService)

{

managerService->HandleDeleteObserver(this, uri);

}

}

You do not have to specify a URI in the HandleDeleteObserver method if you only call managerService->HandleDeleteObserver(this);. All URIs you receive requests for are deleted and you are not listening to any requests anymore.

**Use from a Client perspective**

The following example shows how to send requests from a client perspective:

//Get the microservice

auto \*context = [us::ModuleRegistry::GetModule](http://docs.mitk.org/nightly/classus_1_1ModuleRegistry.html#a99647496a3706320ce624c2bd13da3ea)(1)->[GetModuleContext](http://docs.mitk.org/nightly/classus_1_1Module.html%22%20%5Cl%20%22a1328a9a98dcc3e0c292c8d9bc278373b)();

auto managerRef = context->[GetServiceReference](http://docs.mitk.org/nightly/classModuleContext.html%22%20%5Cl%20%22af47528f02bd276ba9b738b17f60d04f8)<[mitk::IRESTManager](http://docs.mitk.org/nightly/classmitk_1_1IRESTManager.html)>();

if (managerRef)

{

auto managerService = context->GetService(managerRef);

if (managerService)

{

//Call the send request method which starts the actual request

managerService

->[SendRequest](http://docs.mitk.org/nightly/classmitk_1_1IRESTManager.html#a53e817a213a08e555e4def716ac78322)(U("https://jsonplaceholder.typicode.com/posts/1"))

.then([=](pplx::task<web::json::value> resultTask)/\*It is important to use task-based continuation\*/ {

try

{

//Get the result of the request

//This will throw an exception if the ascendent task threw an exception (e.g. invalid URI)

web::json::value result = resultTask.get();

//Do something with the result (e.g. convert it to a QString to update an UI element)

utility::string\_t stringT = result.to\_string();

std::string stringStd(stringT.begin(), stringT.end());

QString stringQ = QString::fromStdString(stringStd);

//Note: if you want to update your UI, do this by using signals and slots.

//The UI can't be updated from a Thread different to the Qt main thread

emit UpdateLabel(stringQ);

}

catch (const [mitk::Exception](http://docs.mitk.org/nightly/classmitk_1_1Exception.html) &exception)

{

//Exceptions from ascendent tasks are catched here

[MITK\_ERROR](http://docs.mitk.org/nightly/mitkLogMacros_8h.html#a5b2267738332420c278264d869c1aa55) << exception.what();

return;

}

});

}

}

The steps you need to make are the following:

1. Get the microservice. You can get the microservice via the module context. If you want to use the microservice within a plug-in, you need to get the module context from [us::ModuleRegistry](http://docs.mitk.org/nightly/classus_1_1ModuleRegistry.html).
2. Call the SendRequest method. This will start the request itself and is performed asynchronously. As soon as the response is sent by the server, the .then(...) block is executed.
3. Choose parameters for .then(...) block. For exception handling, it is important to choose pplx::task<web::json::value> . This is a task-based continuation. For more information, visit <https://docs.microsoft.com/en-us/cpp/parallel/concrt/exception-handling-in-the-concurrency-runtime?view=vs-2017>.
4. Get the result of the request. You can get the JSON-value of the result by calling .get(). At this point, an exception is thrown if something in the previous tasks threw an exception.
5. Do something with the result.

Note

If you want to modify GUI elements within the .then(...) block, you need to do this by using signals and slots because GUI elements can only be modified by the Qt Main Thread. For more information, visit <https://doc.qt.io/Qt-5/thread-basics.html#gui-thread-and-worker-thread>

1. Exception handling. Here you can define the behavior if an exception is thrown, exceptions from ascendant tasks are also caught here.

The code that is followed by this code block shown above will be performed asynchronously while waiting for the result. Besides Get-Requests, you can also perform Put or Post requests by specifying a RequestType in the SendRequest method.

The following example shows how you can perform multiple tasks encapsulated to one joined task. The steps are based on the example for one request and only the specific steps for encapsulation are described.

//Get the microservice

//Get microservice

auto \*context = [us::ModuleRegistry::GetModule](http://docs.mitk.org/nightly/classus_1_1ModuleRegistry.html#a99647496a3706320ce624c2bd13da3ea)(1)->[GetModuleContext](http://docs.mitk.org/nightly/classus_1_1Module.html%22%20%5Cl%20%22a1328a9a98dcc3e0c292c8d9bc278373b)();

auto managerRef = context->[GetServiceReference](http://docs.mitk.org/nightly/classModuleContext.html%22%20%5Cl%20%22af47528f02bd276ba9b738b17f60d04f8)<[mitk::IRESTManager](http://docs.mitk.org/nightly/classmitk_1_1IRESTManager.html)>();

if (managerRef)

{

auto managerService = context->GetService(managerRef);

if (managerService)

{

//Create multiple tasks e.g. as shown below

std::vector<pplx::task<void>> tasks;

for (int i = 0; i < 20; i++)

{

pplx::task<void> singleTask = managerService->[SendRequest](http://docs.mitk.org/nightly/classmitk_1_1IRESTManager.html#a53e817a213a08e555e4def716ac78322)(L"https://jsonplaceholder.typicode.com/posts/1")

.then([=](pplx::task<web::json::value> resultTask) {

//Do something when a single task is done

try

{

resultTask.get();

emit UpdateProgressBar();

}

catch (const [mitk::Exception](http://docs.mitk.org/nightly/classmitk_1_1Exception.html) &exception)

{

[MITK\_ERROR](http://docs.mitk.org/nightly/mitkLogMacros_8h.html#a5b2267738332420c278264d869c1aa55) << exception.what();

return;

}

});

tasks.emplace\_back(singleTask);

}

//Create a joinTask which includes all tasks you've created

auto joinTask = pplx::when\_all(begin(tasks), end(tasks));

//Run asynchonously

joinTask.then([=](pplx::task<void> resultTask) {

//Do something when all tasks are finished

try

{

resultTask.get();

emit UpdateLabel("All tasks finished");

}

catch (const [mitk::Exception](http://docs.mitk.org/nightly/classmitk_1_1Exception.html) &exception)

{

[MITK\_ERROR](http://docs.mitk.org/nightly/mitkLogMacros_8h.html#a5b2267738332420c278264d869c1aa55) << exception.what();

return;

}

});

}

The steps you need to make are the following:

1. Get the microservice. See the example above.
2. Create multiple tasks. In this example, 20 identical tasks are created and are saved into a vector. In general, it is possible to place any tasks in that vector.
3. Do something when a single task is done. Here, an action is performed if a single task is finished. In this example, a progress bar is loaded by a specific number of percent.
4. Create a joinTask. Here, all small tasks are encapsulated in one big task.
5. Run joinTask asynchronously. The then(...) of the joinTask is performed when all single tasks are finished.
6. Do something when all tasks are finished. The handling of the end of a joinTask is equivalent to the end of a single task.