Inhalt

[1. Gegenstand des Dokuments 2](#_Toc413671992)

[2. Allgemeines 3](#_Toc413671993)

[2.1 Logging 3](#_Toc413671994)

[3. Interface Klassen 4](#_Toc413671995)

[3.1 Result Interfaces 4](#_Toc413671996)

[3.2 APIConnector 4](#_Toc413671997)

[3.2.1 Container Methoden des APIConnector 4](#_Toc413671998)

[3.2.2 Der Push 4](#_Toc413671999)

[3.2.3 Agenten Methoden des APIConnector 5](#_Toc413672000)

[3.3 PushCallBack Interface 5](#_Toc413672001)

[3.4 Container Interface 6](#_Toc413672002)

[3.4.1 Container::System Subinterface 7](#_Toc413672003)

[3.4.2 Container::Additional Subinterface 7](#_Toc413672004)

[3.5 Agent Interface 7](#_Toc413672005)

[3.5.1 AgentSetting Interface 8](#_Toc413672006)

[3.6 Interface NotifyMessage 9](#_Toc413672007)

[3.6.1 Interface RSM 9](#_Toc413672008)

[4. Beispiele (teilweise in Form von Pseudo Code) 10](#_Toc413672009)

[4.1 Beispiel zum Erzeugen eines APIKeys 10](#_Toc413672010)

[4.2 Beispiel zum Erzeugen eines Containers 10](#_Toc413672011)

[4.3 Beispiel zum Initialisieren eines Containers 11](#_Toc413672012)

[4.4 Beispiel für den push callback 12](#_Toc413672013)

[4.5 Beispiel zum Erzeugen einer NotifyMessage 12](#_Toc413672014)

# Gegenstand des Dokuments

Diese Dokumentation beschreibt die Klassen und Methoden der C++ Implementierung der ServerEye-API. Sie beschreibt nicht den vollen Umfang der ServerEye-API selbst, da diese Variante nur die Verwendung der API zur Implementierung eigener Container und Agenten abdeckt. Sie beschreibt auch nicht die Funktionsweise von ServerEye selbst. Außerdem ist zu berücksichtigen, dass die API über keine Funktionen zum Löschen von Daten verfügt. D.h. solche Änderungen sind nur über das ServerEye Online Control Center möglich https://occ.server-eye.de. Zusätzlich Information über die API als WEB Interface erhalten sie unter https://api.server-eye.de/.

# Allgemeines

Die Implementierung verwendet den Microsoft Compiler Visual C++ 2013 und dessen Version der Standard Template Library. Die Bibliothek ist als statische multithreaded Bibliothek und mit Unicode Character Unterstützung kompiliert. Die in der API verwendeten Strings sind Unicode Strings. Als Objekt-Klasse wird std::wstring verwendet.

Alle verwendeten Interface Objekte werden ausschließlich unter Verwendung der std::shared\_ptr genutzt. D.h. für jedes Objekt wird immer nur die entsprechende typedef shared\_ptr<TYPE> TYPEPtr; verwendet, so dass kein Objekt mittels delete gelöscht werden sollte. Außerdem wird an keiner Stelle erwartet, dass ein Interface mit new erzeugt wird. Für jedes Objekt wird eine entsprechende statische create Methode zur Verfügung gestellt, solange das Objekt vom Entwickler erzeugt werden muss.

Alle Interfaces liegen im namespace ServerEye.

## Logging

Die API verfügt über Log Mechanismen, die standardmäßig in die Konsole und das Visual Studio Debug Fenster loggen. Vor dem ersten Aufruf einer API Methode kann das Logging auch für Dateien aktiviert werden. Dadurch wird dann ein Logrotate mit ca. 1 MB großen Dateien und einer Plattenbelegung bis zu 100 MB für archivierte Logdateien initialisiert. Dabei wird jedoch sichergestellt dass auf der Partition mindestens 150 MB freier Speicherplatz verbleibt.

Die Dateien enden immer auf \_YYYYMMDD.N.log, N steht für die N-te Datei an diesem Tag. Die Aktivierung erfolgt durch Aufruf von

void Log::enableFileLogging(const wstring& logPath, const wstring& logNamePrefix, const wstring& rotatePath);

z.B.

Log::enableFileLogging(L"D:\\SEAPI.log\\",L"test",L"D:\\SEAPI.log\\logs");

Es werden Logdateien im Verzeichnis D:\SEAPI.log\ mit dem Namen test\_20150325.0.log erzeugt und ins Verzeichnis D:\SEAPI.log\logs rotiert.

Log::enableFileLogging(L"",L"test",L"logs");

Es werden Logdateien im Arbeitsverzeichnis des Programms mit dem Namen test\_20150325.0.log erzeugt und ins Verzeichnis logs im Arbeitsverzeichnis rotiert.

Die Ausgabe von Logmeldungen erfolgt mittels den nachfolgend aufgeführten Log-Methoden. Sie erzeugen die Ausgaben mit den jeweiligen Prefixes:

void Log::Debug(const wstring& message);

void Log::Info(const wstring& message);

void Log::Warn(const wstring& message);

void Log::Error(const wstring& message);

void Log::Fatal(const wstring& message);

Die Ausgaben haben folgendes Format mit vorangestellter Zeit in UTC/GMT.

Jahr/Monat/Tag Stunde:Minute:Sekunde.Microsekunde: <type> Text

2015/03/05 04:49:53.048910: <info> Text

Beispiele:

Log::Info(L"test " + std::to\_wstring(1));

2015/03/05 04:49:53.077910: <info> test 1

Log::Error(L"test " + std::to\_wstring(2));

2015/03/05 04:49:53.084410: <error> test 2

Zu Beginn des Loggings erfolgt immer die Ausgabe des aktuellen Zeitoffsets zu UTC/GMT

2015/03/05 04:49:53.048910: <info> Offset to GMT is 120 min

# Interface Klassen

## Result Interfaces

Alle Ergebnisse von Methoden, die eine Serverabfrage beinhalten, liefern entweder direkt eine Instanz vom Typ Result oder ein davon abgeleitetes Interface, das spezifische Rückgabewerte enthält.

bool Result::getSuccess();

Liefert true, falls die Abfrage erfolgreich ausgeführt wurde.

wstring Result::getMessage();

Liefert im Fehlerfall einen beschreibenden String.

## APIConnector

Der APIConnector ist das Interface zum Aufbau der Verbindung zum Server und um mit diesem Daten auszutauschen.

static void APIConnector::setProxy(const wstring& ip, const wstring& port, const wstring& user, const wstring& password);

Vor der Nutzung der API können die Verbindungsdaten für einen zu verwendenden http Proxy gesetzt werden.

static APIConnectorPtr APIConnector::create();

Erzeugt eine Instanz des Interfaces, die man sich für die weitere Verarbeitung merken muss. Jeder Aufruf erzeugt eine neue Instanz.

CreateKeyResultPtr APIConnector::createKey(const wstring& user, const wstring& password, const wstring& name, const wstring& customerId)

Erzeugt einen neuen API Key für einen Kunden mittels Benutzername und Passwort. Der erzeugte Key kann aus CreateKeyResult::getAPIKey() gelesen werden. Diesen Key sollte man sich lokal speichern (Configfile/Registry) um ihn bei jedem späteren Start wieder verwenden zu können.

void APIConnector::init(const wstring& apiKey);

Setzt den zu verwendeten APIKey für spätere Funktionsaufrufe.

### Container Methoden des APIConnector

ContainerResultPtr APIConnector::getContainer(const wstring& containerId);

Falls man bereits einen Container angelegt hat, so muss man sich die containerId merken und kann mittels dieser den Container wieder vom Server anfordern.

ContainerResultPtr APIConnector::postContainer(const ContainerPtr& container);

Falls man noch keinen Container erzeugt hat, so dient diese Methode zum Anlegen des Containers am Server. Wie ein Container Objekt angelegt wird, siehe Abschnitt Container.

ContainerResultPtr APIConnector::putContainer(const ContainerPtr& container);

Falls sich Containerinformation geändert haben, so dient diese Methode zum Aktualisieren des Containers am Server.

### Der Push

Um Kommandos und Konfigurationen vom Server für einen Container zu erhalten wird eine Push Verbindung benötigt. Von dieser erhält man mittels eines PushCallBack Interface, dass man implementieren muss, Statusinformationen zur Verbindung und muss den Empfang und die Verarbeitung von Nachrichten bestätigen.

void APIConnector::pushStart(PushCallBackPtr& pushCallBack, const ContainerPtr& container)

Initialisiert und startet den Push Service für den gegebenen Container im Hintergrund

void APIConnector::pushStop(const ContainerPtr& container);

Stoppt den Push Service für den gegebenen Container

void APIConnector::pushAck(const wstring& containerId, const wstring& messageId);

Sendet eine Bestätigung während der Verarbeitung einer push Nachricht. Dies muss in Abständen von ca. 25 Sekunden erfolgen, damit der Server den Container nicht als nicht mehr verbunden deklariert. Für kurze Verarbeitungen < ca. 20 Sekunden ist kein pushAck notwendig.

ResultPtr APIConnector::pushResult(const wstring& containerId, const wstring& messageId, bool success);

ResultPtr APIConnector::pushResult(const wstring& containerId, const wstring& messageId, bool success, const wstring& message);

ResultPtr APIConnector::pushResult(const wstring& containerId, const wstring& messageId, bool success, const wstring& message, const wstring& data);

Diese Methoden dienen zum Abschließen einer push Nachricht um dem Server das Ergebnis der Verarbeitung zu übermitteln. Über eine dieser Methoden muss zum Abschluss jedes Callbacks der Empfang und die Verarbeitung bestätigt werden.

### Agenten Methoden des APIConnector

AgentResultPtr APIConnector::getAgent(const wstring& agentId);

Mit dieser Methode wird ein existierender spezifischer Agent vom Server angefordert.

AllAgentsResultPtr APIConnector::getAllAgents(const wstring& containerId);

Mit dieser Methode werden alle Agenten eines Containers vom Server angefordert.

AgentResultPtr APIConnector::postAgent(const AgentPtr& agent);

Mit dieser Methode wird ein Agent auf dem Server erzeugt, falls er noch nicht existiert.

AgentResultPtr APIConnector::putAgent(const AgentPtr& agent);

Mit dieser Methode wird ein Agent auf dem Server aktualisiert, falls er existiert.

NotifyResultPtr APIConnector::notifyAgent(const NotifyMessagePtr& notifyMessage);

Mit dieser Methode werden die Statusinformationen eines Agenten auf dem Server aktualisiert, falls er existiert. Welche Information benötigt werden, muss je nach Anwendungsfall festgelegt werden.

## PushCallBack Interface

Status der Verbindung

void PushCallBack::OnConnect();

Wird aufgerufen, sobald die Verbindung bestätigt wurde.

void PushCallBack::OnDisconnect();

Wird aufgerufen, sobald die Verbindung abbricht oder regulär beim Beenden.

void PushCallBack::OnError();

Wird aufgerufen, sobald die Verbindung abbricht oder wenn ein Fehler oder Timeout auftritt.

void PushCallBack::OnAddOrUpdateAgent(const wstring& containerId, const wstring& senderId, const wstring& control, const wstring& messageId);

Ein Agent wurde hinzugefügt oder aktualisiert. Ob eine Aktion erwartet wird, ist je nach Anwendungsfall zu prüfen.

void PushCallBack::OnDeleteAgent(const wstring& containerId, const wstring& senderId, const wstring& control, const wstring& messageId);

Ein Agent wurde gelöscht. Ob eine Aktion erwartet wird, ist je nach Anwendungsfall zu prüfen.

void PushCallBack::OnRefreshAgent(const wstring& containerId, const wstring& senderId, const wstring& control, const wstring& messageId);

Ein Agent wurde aktualisiert. Ob eine Aktion erwartet wird, ist je nach Anwendungsfall zu prüfen.

void PushCallBack::OnRemoteInstall(const wstring& user, const wstring& password, const wstring& domain, const wstring& containerId, const wstring& control, const wstring& messageId, bool occConnectorCheck, const vector<wstring>& urls);

Eine Remoteinstallation wurde angefordert. Ob eine Aktion erwartet wird, ist je nach Anwendungsfall zu prüfen.

## Container Interface

Das Interface ist designed um einen PC zu repräsentieren. Mittels dieses Interfaces können auch Containerdaten modifiziert werden. Nach dem Setzen einer oder mehrerer Eigenschaften muss man, um dies am Server zu aktualisieren, abschließend die APIConnector::putContainer Methode für diesen Container aufrufen. Nach jedem Systemstart müssen die aktuell am Server gespeicherten Containerdaten abgefragt und aktualisiert werden.

Folgende Typen gibt es

enum ContainerType

{

OCC\_CONNECTOR, // ServerEye spezifisch

SENSORHUB = 2, // ServerEye spezifisch

EXTERNAL = 4 // für Nutzer der API

};

Für Nutzer der API steht nur der Typ EXTERNAL zur Verfügung.

ContainerPtr createContainer(const wstring& customerId, const wstring& name, const wstring& realName, const ContainerType type);

Mit dieser Methode erzeugt man einen Container. Man muss sich die zurückgelieferte Id des Containers merken, damit nicht immer neue Container für gleiche Geräte angelegt werden.

Als Name sollte der Hostname und als realName der vollständige Domänenname angegeben werden.

const wstring& Container::getId() const;

Liefert die UUID des Containers zurück.

const wstring& Container::getRealName() const;

Liefert den einmalig vergebenen Namen des Containers zurück.

const wstring& Container::getCustomerId() const;

Liefert die UUID des Kunden zurück, zu dem dieser Container gehört.

const wstring& Container::getParentId() const;

Liefert die ParentId des Conatinaers zurück. Dies ist nur im Kontext eines SENSORHUB gesetzt.

const wstring& Container::getName() const;

Liefert den Namen des Containers zurück.

int Container::getMaxHeartbeatTimeout() const;

Liefert den aktuell gesetzten Timeout des Containers zurück.

Container::ContainerType Container::getType() const;

Liefert den ContainerType zurück.

void Container::setName(const wstring& name);

Mit dieser Methode kann man den Namen des Containers aktualisieren.

void Container::setMaxHeartbeatTimeout(const int maxHeartbeatTimeout);

Mit dieser Methode kann man den Heartbeattimeout des Containers setzen.

Container::System& Container::getSystem();

const Container::System& Container::getSystem() const;

Mit diesen Methoden kann auf die Systemeigenschaften des Containers zum Modifizieren bzw. Auslesen (const) zugegriffen werden.

Additional& Container::getAdditional();

const Additional& Container::getAdditional() const;

Mit diesen Methoden kann auf die Zusatzdaten des Containers zum Modifizieren bzw. Auslesen (const) zugegriffen werden.

### Container::System Subinterface

const wstring& Container::System::getIp() const;

void Container::System::setIp(const wstring& ip);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten IP Adresse.

bool Container::System::isServer() const;

void Container::System::setServer(const bool value);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten Servereigenschaft.

bool Container::System::isVM() const;

void Container::System::setVM(const bool value);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten VirtualMachine-Eigenschaft.

int Container::System::getNumberOfProcessors() const;

void Container::System::setNumberOfProcessors(const int processors);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten Anzahl der Prozessoren.

long long Container::System::getTotalRam() const;

void Container::System::setTotalRam(const long long totalRam);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten Größe des Arbeitsspeichers.

int Container::System::getArchitecture() const;

void Container::System::setArchitecture(const int architecture);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten Architektur der CPU.

### Container::Additional Subinterface

const wstring& Container::Additional::getName() const;

void Container::Additional::setName(const wstring& name);

Methoden zum Modifizieren oder Auslesen des im Container gespeicherten Namens des Betriebssystems.

const wstring& Container::Additional::getVersion() const;

void Container::Additional::setVersion(const wstring& version);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten Version des Betriebssystems.

const wstring& Container::Additional::getServicePack() const;

void Container::Additional::setServicePack(const wstring& servicePack);

Methoden zum Modifizieren oder Auslesen der im Container gespeicherten Version des verwendeten Servicepacks des Betriebssystems.

long long Container::Additional::getLastStart() const;

void Container::Additional::setLastStart(const long long lastStart);

Methoden zum Modifizieren oder Auslesen des im Container gespeicherten Zeitpunkts des letzten Systemstarts in Millisekunden seit dem 1.1.1970 in UTC.

## Agent Interface

Agenten sind anwendungsspezifisch zu definieren. Das Interface stellt nur allgemeine Zugriffsmethoden zur Verfügung. Sie bestehen aus wenigen Attributen und einem Satz aus Einstellungen, je nach Anwendungsfall.

static AgentPtr Agent::createAgent(const wstring& parentId, const wstring& name, const wstring& type);

Mit dieser Methode erzeugt man einen Agenten. Nach dem Setzen einer oder mehrerer Eigenschaften muss man, um dies am Server zu aktualisieren, abschließend die APIConnector::putAgent Methode für diesen Container aufrufen. Um Agenten nicht doppelt anzulegen, muss man nach dem Abfragen des Containers die Liste alle Agenten abfragen und diese mit den benötigten Agenten abgleichen.

const wstring& Agent::getParentId() const;

Liefert die Id des Containers zurück.

const wstring& Agent::getId() const;

Liefert die Id des Agenten zurück.

const wstring& Agent::getName() const;

Liefert die Namen des Agenten zurück.

const wstring& Agent::getType() const;

Liefert den Typ des Agenten zurück. Aktuell unterstützte Typen sind:

const static wstring Agent::TYPE\_EXTERNAL\_HARDWARE; // "EXTERNAL\_HARDWARE"

const static wstring Agent::TYPE\_EXTERNAL\_SOFTWARE; // "EXTERNAL\_SOFTWARE"

const static wstring Agent::TYPE\_EXTERNAL\_NETWORK; // "EXTERNAL\_NETWORK"

void Agent::setName(const wstring& name);

Mit dieser Methode kann man den Namen des Containers aktualisieren.

const Agent::SettingMap& Agent::getAllSettings() const;

Liefert alle Einstellung in Form einer Map zurück in der alle Einstellung über Schlüssel gespeichert sind.

bool Agent::hasSetting(const wstring& key) const;

Liefert true, wenn eine Einstellung mit Schlüssel key vorhanden ist.

AgentSettingPtr Agent::getSetting(const wstring& key) const;

Liefert die Einstellung für den Schlüssel key zurück oder AgentSettingPtr isNull.

void Agent::setSetting(const AgentSettingPtr& setting);

Methode zum Aktualisieren oder Setzen von Einstellungen.

bool Agent::removeSetting(const wstring& key);

Methode zum Entfernen der Einstellungen mit dem Schlüssel key.

### AgentSetting Interface

Agenten Einstellungen sind anwendungsspezifisch und deshalb dynamisch gehalten. Sie bestehen aus Schlüssel/Wert Paaren mit zusätzlichen Werten, die die Anzeige und die Editierbarkeit regeln.

static AgentSettingPtr AgentSetting::createSetting(const wstring& key);

Mit dieser Methode erzeugt man ein AgentSetting.

const wstring& AgentSetting::getKey() const;

Liefert den Schlüssel zurück.

const wstring& AgentSetting::getValue() const;

Liefert den Wert zurück.

int AgentSetting::getOrderInt() const;

Liefert die Anzeige-Ordnungs-Id zurück. Darüber wird die Reihenfolge in der Anzeige bestimmt.

bool AgentSetting::getEditable() const;

Liefert das Editierbar-Flags zurück.

const wstring& AgentSetting::getDefinition() const;

Liefert die Editordefinition in json Format. // Beispiel: { 'type' : 'string', 'allowBlank' : false }

void AgentSetting::setValue(const wstring& value);

Methode zum Setzen des Werts.

void AgentSetting::setOrderId(const int orderId);

Methode zum Setzen der Anzeige-Ordnungs-Id. Darüber wird die Reihenfolge in der Anzeige bestimmt.

void AgentSetting::setEditable(const bool value);

Methode zum Setzen der Editierbar-Flags.

void AgentSetting::setDefinition(const wstring& definition);

Methode zum Setzen der Editordefintion in json Format. // Beispiel: { 'type' : 'string', 'allowBlank' : false }

## Interface NotifyMessage

Die NotifyMessage wird verwendet, um die Agenten Status Informationen am Server zu aktualisieren. Die benötigten Informationen sind anwendungsspezifisch festzulegen.

NotifyMessagePtr NotifyMessage::createNotifyMessage(const wstring& agentId, const wstring& parentId, bool error, const wstring& message, const RSMPtr rsm, const long long utcMs);

NotifyMessagePtr NotifyMessage::createNotifyMessage(const wstring& agentId, const wstring& parentId, bool error, const wstring& message, const long long utcMs);

NotifyMessagePtr NotifyMessage::createNotifyMessage(const wstring& json);

Statische Methoden zum Erzeugen von NotifyMessages.

Die Details der Nachricht sind mit Krämer-IT abzuklären.

Die parentId ist in der Regel der Container des Agenten.

Die Zeit der Nachricht utcMs ist in Millisekunden seit dem 1.1.1970 in UTC anzugeben.

### Interface RSM

Die ist ein Hilfsinterface um die Detaildaten zur Notifymessage zu erzeugen, die im OCC via Template abgebildet werden. Die benötigten Informationen sind anwendungsspezifisch festzulegen.

static const RSMPtr RSM::createRSM(const wstring& key, const int version, const State state, const wstring& data);

Statische Methoden zum Erzeugen des RSM Objekts.

Die Details des Objects sind mit Krämer-IT abzuklären.

State ist als enum in der Klasse definiert und beschreibt Zustände wie RSM::OK oder RSM::Error.

# Beispiele (teilweise in Form von Pseudo Code)

Die Beispiele gehen davon aus, dass eine Variable APIConnectorPtr apiConnector = APIConnector::create(); existiert und ein benötigter Proxy bereits gesetzt ist

## Beispiel zum Erzeugen eines APIKeys

// Get full domaine name

wstring fullDomainName = MyTools::getFullDomaineName();

if (fullDomainName.size())

{

Log::Info(L"Computer name is " + fullDomainName);

}

else

{

Log::Error(L"Could not resolve full domaine name");

return CMD\_FAILED;

}

// Create an API key with the ServerEye API

CreateKeyResultPtr result = apiConnector->createKey(user, password, fullDomainName, customerId);

if (!result->getSuccess())

{

Log::Error(L"Could not create API key: " + result->getMessage());

return CMD\_FAILED;

}

// Get the key and store it for later use

wstring apiKey = result->getAPIKey();

## Beispiel zum Erzeugen eines Containers

// Create a new container

wstring name = MyUtils::getHostName();

wstring realName = MyUtils::getFullDomaineName();

ContainerPtr container = Container::createContainer(customerId, name, realName, Container::EXTERNAL);

// update the container information

MyTools::updateContainerData(container);

// submit to server

auto result = apiConnector->postContainer(pContainer);

if (!result->getSuccess())

{

Log::Error(L"Could not create Container: " + result->getMessage());

return CMD\_FAILED;

}

// Get the container and store it for later use

containerId = result->getData()->getId();

## Beispiel zum Initialisieren eines Containers

auto containerResult = apiConnector->getContainer(m\_ContainerId);

if (!containerResult->getSuccess())

{

return FAILED;

}

// update the current container information

ContainerPtr container = containerResult->getData();

MyTools::updateContainerData(container);

// put the updated container to OCC

ContainerResultPtr result = apiConnector->putContainer(container);

if (!result->getSuccess())

{

Log::Error(L"Container update failed: " + result->getMessage());

return FAILED;

}

else

{

Log::Info(L"Container update successfull");

}

AllAgentsResultPtr allAgents = apiConnector->getAllAgents(containerId);

if (allAgents->getSuccess())

{

Log::Debug(L"Searching for existing agents in " + containerId);

AgentList agents = allAgents->getData();

// compare agents with exiting by

// (mySensor->getName() == pAgent->getName() &&

// mySensor->getSensorType() == pAgent->getType())

// set the settings delivered by the server

// e.g. mySensor->setSetting(pAgent->getAllSettings());

// for no longer used Agents send a Notification to show an error,

// e.g. for a no longer existing harddisk

}

// if there are agents missing create them with

// apiConnector->postAgent(Agent::createAgent(containerId, name, typeId))

// if this is all done you are ready to start the push

apiConnector->pushStart(myCallback, containerResult->getData());

## Beispiel für den push callback

Aktuell besteht nur die Notwendigkeit auf das OnAddOrUpdateAgent Callback zu reagieren. Alle anderen dienen nur zur Anzeige bzw. sind anwendungsspezifisch zu berücksichtigen.

void MyCallback::OnAddOrUpdateAgent(const wstring& containerId, const wstring& senderId, const wstring& control, const wstring& messageId)

{

// Add or update the specific agent

if (myContainerId == containerId)

{

AgentResultPtr agentResult = apiConnector->getAgent(senderId);

if (agentResult->getSuccess())

{

// Check for a local sensor bound to this OCC agent

for (MySensor pSensor : mySensors)

{

if (pSensor->getAgentId() == senderId)

{

// We found the sensor which is bound to the server agent

bool result = pSensor->setSettings(agentResult->getData()->getAllSettings());

apiConnector->pushResult(myContainerId, messageId, result);

break;

}

}

}

else

{

Log::Error(L"Updating agent failed: " + agentResult->getMessage());

}

}

else

{

Log::Warn(L"MonitoringManager OnAddOrUpdateAgent() failed: ContainerId does not match");

}

}

## Beispiel zum Erzeugen einer NotifyMessage

bool error;

wstring messageText;

wstring rsm\_jsonData;

... fill your agent data

// Build the notification message for the server

NotifyMessagePtr notifyMessage = NotifyMessage::createNotifyMessage(

agentId,

containerId,

error,

messageText,

RSM::createRSM(L"main", 1, (error ? RSM::OK : RSM::Error), rsm\_jsonData),

myUtils::getTime());

apiConnector->notifyAgent(notifyMessage);