Grenton Gate Http

Kompleksowa integracja z systemami zewnętrznymi przy użyciu urządzenia Grenton Gate Http

Wersja dokumentu: 1.0

Data: 9.09.2019

Gate Http: Firmware w wersji 1.0.1 (1932) lub wyższej

Opis konfiguracji konfiguracji krok po kroku na przykładzie wyjścia przekaźnikowego.

System

Powiedzmy, że mamy prosty system złożony z następujących elementów:

- CLU Z-Wave - nazwany (Name) "CluZ"
- Moduł przekaźnikowy - na potrzeby przykładu użyjemy jednego wyjścia o nazwie "Relay"
- Gate Http - nazwa "GateHttp"

Sterowanie wyjściem

W celu umożliwienia sterowania wyjściem przekaźnikowym z zewnętrznego systemu tworzymy nowy obiekt typu HttpListener na GateHttp i konfigurujemy jak poniżej:

- Name: RelayControlListener
- Path: /relaycontrol

Pozostałe parametry pozostawiamy na razie bez zmian.

Skrypt

Aby obiekt RelayControlListener zadziałał należy utworzyć dla skrypt, który będzie obsługiwał przychodzące zapytania Http.

Tutaj warto zauważyć, że z tego skryptu mamy dostęp do całego systemu i wszystkich jego funkcjonalności. To otwiera praktycznie nieograniczone możliwości ale też rodzi pewne ryzyka, zwłaszcza jeśli funkcjonalność Gate'a nie jest dobrze przemyślana. Dlatego zwracamy szczególną uwagę, że implementując funkcjonalność Gate'a należy dobrze przemyśleć sposób działania jaki chcemy osiągnąć oraz to jak działanie Gate'a może zależeć albo wpływać na inne elementy systemu. Przykłady takiego podejścia będą też omawiane dalej.

Wracając do skryptu kontrolującego Relay. Chcemy mieć możliwość załączenia lub rozłączenia Relay'a wysyłając do niego oczekiwany stan (On/Off) albo wykonania metody switch. Takie podejście do implementacji umożliwa podłączenie do niego zarówno kontroli typu przełącznika bistabilnego jak i monostabilnego.

Przechodząc do działania, tworzymy na GateHttp skrypt o nazwie RelayControlOnRequest, i w trybie edycji kodu wrzucamy to co poniżej:

```
-- RelayControlOnRequest()
local data = GateHttp->RelayControlListener->QueryStringParams
if data == nil then
  CluZ->Relay->Switch(0)
else
  if data.cmd == "setValue" then
    local val = tonumber(data.val)
    if(val == 1) then
      CluZ->Relay->SwitchOn(0)
    elseif(val == 0) then
      CluZ->Relay->SwitchOff(0)
    end
  end
end

GateHttp->RelayControlListener->StatusCode = 200
GateHttp->RelayControlListener->ResponseBody = "OK"
GateHttp->RelayControlListener->SendResponse()
```

Natępnie przypisujemy skrypt do zdarzenia OnRequest obiektu RelayControlListener i wysyłamy konfigurację do systemu.

Powyższy skrypt pobiera z obiektu RelayControlListener wartości parametrów zapytania i zależnie od tego co w nich się znajduje wykonuje odpowiednie akcje. Po czym odsyła do klienta status operacji - w tym przypadku 200, OK.

Działanie można łatwo przetestować za pomocą zwykłej przeglądarki internetowej wpisując poniższe URLe (oczywiście adres IP należy zamienić na rzeczywisty adres Waszego Gate's Http):

`http://192.168.88.4/relaycontrol?cmd=setValue&val=1` - Włącza Relay

`http://192.168.88.4/relaycontrol?cmd=setValue&val=0` - Wyłącza Relay

`http://192.168.88.4/relaycontrol` - Przełącza stan Relay'a

Jak widać na przykładach możemy użyć listenera na dwa sposoby. Jeśli parametr "cmd" (komenda do wykonania) i "val" (wartość do ustawienia) są odpowiednio zdefiniowane to ustawiają konkretny stan Relay'a. Jeśli pominiemy w URL'u te parametry to obiekt działa jak przełącznik (switch).

Powyższy przykład może zostać dalej rozbudowany o kolejne komendy jeśli jest potrzeba wykonania innych komend. Można też dodać kolejne parametry identyfikujące obiekt na którym te komendy należy wykonać.

Pobieranie stanu

W poprzednim kroku umożliwiliśmy sterowania obiektem w systemie z zewnątrz. Bardzo często w kolejnym kroku pojawia się potrzeba udostępnienia także możliwości pobrania aktualnego stanu obiektu.

Jedną z szybszych i najbardziej intuicyjnych metod (niekoniecznie najlepszą) jest zdefiniowane kolejnego Listenera który pobierze wartość Value z obiektu Relay i odeśle do klienta. Najprostrzy skrypt relizujący taką funkcjonalność wyglądać może jak poniżej.

```
-- RelayStateOnRequest()
GateHttp->RelayState->StatusCode = 200
GateHttp->RelayState->ResponseBody = "Relay State: "..CluZ->Relay->Value
GateHttp->RelayState->SendResponse()
```

Wpisując poniższy URL do przeglądarki widać, że dostajemy odpowiedź ze stanem obiektu Relay (w prostej postaci tekstowej, ale to nie format przysłania danych jest tematem tego przykładu).

`http://192.168.88.4/relaystate` - zwraca "Relay State: 0" lub "Relay State: 1" zależnie od stanu obiektu.

Powyższy przykład na pierwszy rzut oka działa dobrze ale spróbujmy się przyglądnąć bliżej.

Kolejność zdarzeń

Skonstruowaliśmy właśnie interfejs (API) Http dwie metody:

- /relaycontrol - umożliwia kontrolowanie obiektem Relay
- /relaystate - zwraca aktualny stan (wartość) obiektu Relay

Po szybkim przetestowaniu wszystko działa dobrze ale jak pisaliśmy wyżej należy jeszcze zastanowić się jak takie metody zostaną użyte. Mianowicie, łatwo sobie wyobrazić, że w zewnętrznym systemie te dwie metody zostaną użyte tuż po sobie: wywołanie akcji przełączenia oraz po otrzymaniu odpowiedzi, odczytanie stanu w celu potwierdzenia, że akcja nastąpiła i synchronizacji statusu.

I tutaj może nastąpić nastąpić nieoczekiwane działanie sytemu, Relay się załącza ale zwracany stan jest 0, czyli nieprawidłowy. Przyczyną takiego stanu rzeczy jest fakt, że operacje te są wykonywane asynchronicznie na dwóch różnych urządzeniach. Nie ma gwarancji, że operacja zmiany stanu Relay'a zdąży się wykonać zanim zapytamy o jego stan. Wywołanie akcji zmiany stanu w skrypcie RelayControlOnRequest() wywoływane jest asynchronicznie, co oznacza, że skrypt nie czeka aż CluZ wykona zadanie.

Rozważany przypadek jest bardzo prosty i praktycznie zawsze zadziała ale w przypadku bardziej skomplikowanych operacji (gdy zaangażowane są różne obiekty celu, do wykonania operacji konieczna jest jeszcze wymiana danych, przesłanie cech, itd) ryzyko, że pobranie statusu nastąpi przed realną zmianą stanu obiektu(ów) jest realny i w złożonych systemach często obserwujemy takie efekty.

Synchronizacja zdarzeń

Powyższy problem można rozwiązać zmuszając skrypt RelayControlOnRequest() aby zaczekał, aż CluZ realnie wywoła na docelowym urządzeniu akcję zmiany stanu. Można to w prosty sposób osiągnąć za pomocą funkcji clu.await(). Np.: wywołanie:

`CluZ->Relay->Switch(0)`

zamieniamy na:

`clu.await(CluZ->Relay->Switch(0))`

(Pozostałe wywołania akcji na CluZ zamieniamy w analogiczny sposób).

Od teraz nasz Listener nie wyśle potwierdzenia "200, OK" zanim akcja na CluZ nie zostanie realnie wykonana więc klient korzystający z tego interfejsu nie zostanie wprowadzony w błąd poprzez zbyt szybkie potwierdzenie wykonania zadania.

Funkcja clu.await() ma jednak pewne ograniczenie. Limit czasowy na wykonanie wywołania jest 800ms o jeśli nie uda się zrealizować zadania w takim czasie skrypt zakończy się timeout'em i klient Http dostanie w odpowiedzi błąd Http: "500 Internal Server Error".

W większości przypadków takie timeout nie jest problemem i system będzie działał poprawnie ale w przypadku złożonych operacji i/lub gdy CluZ będzie obciążone innymi zadaniami może się to wydarzyć. Sposób na rozwiązanie problemu w takim przypadku został opisany w kolejnej sekcji.

Potwierdzenie zwrotne

W złożonych systemach oraz tam gdzie zależy nam na dużej niezawodności i stabilności działania integracji należy opóźnić odpowiedź Listenera Http do czasu otrzymania wprost potwierdzenia z CluZ, że zadanie zrealizowano.

W tym celu rozbijamy działanie na dwa etapy. Zamiast jednego skryptu realizującego całość zadania definiujemy dwa: pierwszy realizuje zadanie, drugi wysyła odpowiedź Http po potwierdzeniu przez CluZ, że zakończono wykonywanie zadania.

Dla przejrzystości definiujemy nowy skrypt i przypisujemy go do Listenera:

- Event OnRequest: GateHttp->SplitSyncOnRequest()

Skrypt SplitSyncOnRequest() wygląda następująco:

```
-- SplitSyncOnRequest()
local data = GateHttp->RelayControlListener->QueryStringParams
if data == nil then
  CluZ->SplitSyncCluzTask("Switch")
  return
else
  if data.cmd == "setValue" then
    local val = tonumber(data.val)
    if(val == 1) then
      CluZ->SplitSyncCluzTask("On")
      return
    elseif(val == 0) then
      CluZ->SplitSyncCluzTask("Off")
      return
    end
  end
end

GateHttp->RelayControlListener->StatusCode = 400
GateHttp->RelayControlListener->ResponseBody = "Bad request"
GateHttp->RelayControlListener->SendResponse()
```

W każdym miejscu skryptu gdy delegujemy zadnie do CluZ (tym razem poprzez dodatkowy skrypt o czym za chwilę) kończymy działanie naszego skryptu, bez wysyłania odpowiedzi Http do klienta. Jeśli wykonanie skryptu dojdzie do końcowych linii będzie to oznaczać, że nie udało się prawidłowo zinterpretować zapytania co oznacza, że jest ono nieprawidłowo i odsyłamy błąd "400, Bad request". Przy okazji dodaliśmy kolejny poziom zabezpieczenia przed nieprawidłowymi parametrami wywołania.

Zadanie nie jest teraz jak poprzednio wykonywane bezpośrednio na docelowym obiekcie Relay ale delegowane do skryptu na CluZ o nazwie SplitSyncCluzTask(action: string). Zastosowana notacja oznacza, że skrypt jest wywoływany z parametrem o nazwie "action", który jest typu "string" - nie mylić z notacją LUA gdzie nie definiujemy typu parametru wywołania funkcji. Parametr "action" definiuje konkretną akcję do wywołania na obiekcie Relay. Działanie jest identyczne jak w poprzednim przypadku.

```
-- SplitSyncCluzTask(action: string)
if(action == "On") then
  CluZ->Relay->SwitchOn(0)
elseif (action == "Off") then
  CluZ->Relay->SwitchOff(0)
elseif (action == "Switch") then
  CluZ->Relay->Switch(0)
else
  -- Unknown action
  GateHttp->SplitSyncRequestCompleted(false)
  -- Return to avoid double completion
  return
end
GateHttp->SplitSyncRequestCompleted(true)
```

Zależnie od zdefiniowanej akcji wykonywana jest odpowiednia metoda na obiekcie Relay. Na końcu informujemy GateHttp, że zakończyliśmy zadanie i należy odesłać do klienta odpowiedź. Została do tego celu stworzona metoda na GateHttp o nazwie SplitSyncRequestCompleted(success: boolean), która jako parametr przyjmuje wartość logiczną: true jeśli udało się wykonać akcję, false w przeciwnym przypadku.

```
-- SplitSyncRequestCompleted(success: boolean)
if success then
  GateHttp->RelayControlListener->StatusCode = 200
  GateHttp->RelayControlListener->ResponseBody = "OK"
else
  GateHttp->RelayControlListener->StatusCode = 405
  GateHttp->RelayControlListener->ResponseBody = "Not allowed"
end
GateHttp->RelayControlListener->SendResponse()
```

GateHttp poprzez powyższą metodę wysyła odpowiedź do klienta informującą o sukcesie lub błędzie zależnie od otrzymanego parametru. Tym sposobem zrealizowaliśmy w pełni synchrorniczną metodę Http, która nie ma ograniczenia czasowego na działanie. W bardziej zaawansowanych przypadkach można jeszcze bardziej usprawnić działanie systemu poprzez wywołanie funkcji

SplitSyncRequestCompleted(success: boolean) w odpowiedzi na zdarzenia informujące o zmianie wartości konkretnego obiektu. Co daje pewność, że zmiana nastąpiła i jeszcze bardziej zwiększa stabilność systemu.

W przypadku danych otrzymywanych z zewnętrznych systemów należy zawsze stosować metodę ograniczonego zaufania co do ich poprawności. Zalecamy nie przekazywanie bezpośrednio wartości do metod i skryptów wewnątrz systemu a stosowanie konkretnych akcji w zależności od wartości metod jak widać w powyższych skryptach. W przypadku konieczności bezpośredniego użycia zmiennych otrzymanych z zewnątrz należy je przekazywać za pośrednictwem zmiennych użytkownika (które są adresowalne w całym systemie Grenton i można je swobodnie przesyłać między urządzeniami CLU). Dodatkowo każdą zmienną otrzymaną z zewnątrz należy w skrypcie walidować pod kątem jest poprawności, wartości, zakresu. Brak odpowiedniej weryfikacji otrzymanych wartości może powodować nieoczekiwane działanie systemu, otworzyć dostęp do niechcianych funkcjonalności a nawet powodować błędy oraz przejście CLU w tryb emergency.

Timeout

Stoworzony Listener działa już prawie niezawodnie. Dlaczego prawie? Zastanówmy się co się stanie jeśli CluZ z jakiegoś powodu nigdy nie wywoła metody SplitSyncRequestCompleted(success: boolean). GateHttp zostaje wtedy w stanie oczekiwania na zakończenie obsługi bieżącego zapytania i przestaje reagować na kolejne zapytania.

Oczywiście w dobrze skonfigurowanym systemie nie powinno się to zdarzyć. Jednak zawsze może wystąpić niespodziewana sytuacja i dlatego każdy element systemu powinien być skonfigurowany tak aby działał możliwie niezależnie i był odporny na błędy w innych obszarach. Dlatego nasz Listener też powinien być w pełni odporny na takie sytuacje.

W tym celu na GateHttp zdefiniujemy obiekt Timer, który będzie pilnował, żeby oczekiwanie na odpowiedź CluzZ nie trwała w nieskończoność. Parametry nowego obiektu:

- Name: SplitSyncTimeout
- Event OnTimer: GateHttp->SplitSyncTimeoutOnTimer()
- Time: 3000 - tutaj należy dobrać czas odpowiednio do konkretnej sytuacji, na potrzeby przykładu przyjmujemy 3s (3000ms)
- Mode: CountDown

Skrypt wykonywany po upłynięciu zadanego czasu wygląda następująco:

```
-- SplitSyncTimeoutOnTimer()
GateHttp->RelayControlListener->StatusCode = 408
GateHttp->RelayControlListener->ResponseBody = "Timeout"
GateHttp->RelayControlListener->SendResponse()
```

Działanie skryptu jest dość proste, po prostu zwraca błąd "408, Timeout".

Żeby wszystko zadziałało należy odpowiednio zmodyfikować skrypty SplitSyncOnRequest() oraz SplitSyncRequestCompleted(success: boolean).

```
-- SplitSyncOnRequest()
```

```
local data = GateHttp->RelayControlListener->QueryStringParams
if data == nil then
  CluZ->SplitSyncCluzTask("Switch")
  GateHttp->SplitSyncTimeout->Start()
  return
else
  if data.cmd == "setValue" then
    local val = tonumber(data.val)
    if(val == 1) then
      CluZ->SplitSyncCluzTask("On")
      GateHttp->SplitSyncTimeout->Start()
      return
    elseif(val == 0) then
      CluZ->SplitSyncCluzTask("Off")
      GateHttp->SplitSyncTimeout->Start()
      return
    end
  end
end

GateHttp->RelayControlListener->StatusCode = 400
GateHttp->RelayControlListener->ResponseBody = "Bad request"
GateHttp->RelayControlListener->SendResponse()
```

Za każdym razem gdy delegujemy zadanie do CluZ startujemy timer SplitSyncTimeout.

```
-- SplitSyncRequestCompleted(success: boolean)
if(GateHttp->SplitSyncTimeout->State == 1) then
  GateHttp->SplitSyncTimeout->Stop()
  if success then
    GateHttp->RelayControlListener->StatusCode = 200
    GateHttp->RelayControlListener->ResponseBody = "OK"
  else
    GateHttp->RelayControlListener->StatusCode = 405
    GateHttp->RelayControlListener->ResponseBody = "Not allowed"
  end
  GateHttp->RelayControlListener->SendResponse()
end
```

Natomiast w skrypcie SplitSyncRequestCompleted(success: boolean) sprawdzamy najpierw czy timer jest ciągle w stanie "1" (włączonym). Zapobiega to przed niepotrzebną próbą wysyłania odpowiedzi w sytuacji gdy timeout już wystąpił - czas na odpowiedź się skończyć i została wysłana odpowiedź informująca o wystąpieniu błędu "408, Timeout". Jeśli timer ciągle działa (normalna sytuacja, czas na odpowiedź się nie wyczerpał) zatrzymujemy timer i dalej postępujemy tak jak poprzednio.

Wiele obiektów

Wróćmy jeszcze na chwilę do metody pobierającej stan Relay'a. W szczególności przyjrzyjmy się jeszcze

raz następującej linii:

```
GateHttp->RelayState->ResponseBody = "Relay State: "..CluZ->Relay->Value
```

Kluczowe tutaj jest pobranie wartości cechy Value obiektu Relay:

```
CluZ->Relay->Value
```

Ta metoda działa dobrze ale należy zdawać sobie sprawę, że wartość tej cechy pobierana jest w momencie wykonywania skryptu. W jej wyniku następuje komunikacje między GateHttp a CluZ przez sieć. Jest to wywołanie synchroniczne, czyli metoda czeka aż zostanie dostarczona odpowiedź z wartością cechy Value obiekty Relay. Wiemy już o pewnych ograniczeniach takiego wywołania. W tym konkretnym przypadku zagrożeń jest jeszcze więcej. Mianowicie, wartość tej cechy jest pobierana za każdym razem gdy klient zapyta o jej wartość przez interfejs Http co generuje niepotrzebny ruch w systemie. Dodatkowo wprowadza niepotrzebne opóźnienie w systemie. Jeśli takich zapytań jest dużo to może to mieć wpływ na wydajność systemu. W niektórych zwłaszcza prostych przypadkach jest to akceptowalne i system będzie sobie z tym dobrze radził. Ale nie zawsze.

Wyobraźmy sobie, że w systemie jest wiele obiektów i potrzebujemy w odpowiedzi dostarczyć statusy wszystkich z nich (w postaci JSON lub CSV). Jeśli w takim przypadku zastosujemy identyczną metodę to skrypt realizujący takie zadanie może wyglądać mniej więcej jak poniżej:

```
GateHttp->RelayState->StatusCode = 200

local response = CluZ->Relay01->Value
response = response .. "," .. CluZ->Relay02->Value
response = response .. "," .. CluZ->Relay03->Value
response = response .. "," .. CluZ->Relay04->Value
response = response .. "," .. CluZ->Relay05->Value
response = response .. "," .. CluZ->Relay06->Value
response = response .. "," .. CluZ->Relay07->Value
response = response .. "," .. CluZ->Relay08->Value
response = response .. "," .. CluZ->Relay09->Value
response = response .. "," .. CluZ->Relay10->Value
response = response .. "," .. CluZ->Relay11->Value

GateHttp->RelayState->ResponseBody = "System State: ".. response
GateHttp->RelayState->SendResponse()
```

W rzeczywistym systemie obiektów relay może być znacznie więcej. Każda linia powoduje odpytanie do CluZ o wartość cechy Value przez sieć. Zebranie stanu wszystkich obiektów może zabrać sporo czasu. Opóźnia to znacznie odpowiedź i na czas wykonywania operacji blokuje GateHttp.

Seria odpytań występuje każdorazowo gdy klient zapyta o stan systemu. W większości przypadków wartość cechy między zapytaniami zmienia się tylko dla jednego obiektu, tego właśnie zmienionego. Te wszystko powoduje dużo niepotrzebnego ruchu i negatywnie wpływa na szybkość działania systemu. Z punktu widzenia użytkownika końcowego system może w takich przypadkach działać niestabilnie, mieć nieoczekiwane opóźnienia, zawieszać się na krótkie lub dłuższe chwile a nawet gubić niektóre

zdarzenia.

Stan dla złożonego systemu

W celu rozwiązania powyższego problemu należy trochę inaczej podejść do zadania pobierania stanu urządzeń. W dalszej części tej sekcji, dla prostoty przykładów, wrócimy do pojedynczego obiektu Relay, ale podana metoda zadziała praktycznie dla dowolnej liczby obiektów.

Powiedzmy, że zamiast odpytywać zdalne CluZ o stan obiekty Relay za każdym razem kiedy klient o niego zapyta moglibyśmy trzymać jego wartość lokalnie w zmiennej użytkownika GateHttp. Dzięki temu kiedy klient zapyta bez żadnych opóźnień zwracamy jej wartość natychmiast bez żadnych opóźnień, nazwijmy ją RelayValueOnGateHttp. Co więcej chcielibyśmy wyeliminować wszystkie zapytania synchronizujące jej wartość i dostawać informacje tylko wtedy kiedy jest to potrzebne, czyli kiedy wartość cechy CluZ->Relay->Value się zmieni. Aby to osiągnąć do zdarzenia OnValueChange obiektu Relay przypisujemy następującą komendę:

`GateHttp->RelayValueOnGateHttp=CluZ->Relay->Value`

Co mniej więcej oznacza: Za każdym razem kiedy wartość cechy Value się zmieni, przypisz do cechy użytkownika RelayValueOnGateHttp na GateHttp tą nową wartość. Od teraz po stronie GateHttp zawsze będziemy mieli aktualną wartość obiektu Relay. W momencie zapytania wysyłamy po prostu tą wartość do klienta. Aby to zrealizować modyfikujemy skrypt RelayStateOnRequest() w następujący sposób:

```
-- RelayStateOnRequest()
GateHttp->RelayState->StatusCode = 200
GateHttp->RelayState->ResponseBody = "Relay State: "..GateHttp-
>RelayValueOnGateHttp
GateHttp->RelayState->SendResponse()
```

Jak wspomniano wcześniej można zastosować dla dowolnie wielu obiektów i nie powoduje żadnego negatywnego wpływu na wydajność systemu ponieważ komunikowane są tylko zmiany poszczególnych wartości w momencie kiedy wystąpią.

Push Notyfikacje

Idąc krok dalej na drodze do idealnej integracji zaimplementujmy jeszcze jedno usprawnienie. Jak dotąd klient sam musiał dopytywać co chwilę czy przypadkiem coś nie zmieniło się w systemie. Jeśli system ma być responsywny to takie zapytania muszą odbywać się często. Częste zapytania generują niepotrzebny ruch i zwiększają ryzyko opóźnień, zwłaszcza w obsłudze zdarzeń bardzo wrażliwych na opóźnienia jak np.: włączanie oświetlenia, gdzie użytkownik od razu czuje, że akcja nie nastąpiła natychmiast po dotknięciu przycisku.

Dodatkowo klient nie jest notyfikowany natychmiast a zmianie w systemie ale dopiero w momencie kiedy sam dopyta czy aby nic się nie zmieniło.

Rozwiązaniem jest metoda Push stanu gdzie to system sam aktywnie wysyła notyfikację a zmianie stanu urządzenia w systemie. W celu zaimplementowania takiego mechanizmu tworzymy nowy obiekt na GateHttp typu HttpRequest:

- Name: StatePushNotification
- Host: IP:Port serwera http nasłuchującego informacji o zmianach stanu
- Path: /statechanged
- Method: PUT

Pozostałe ustawienia bez zmian.

Następnie dodajemy nowy skrypt SendStatePushNotification(newValue: number):

```
-- SendStatePushNotification(newValue: number)
GateHttp->StatePushNotification->SetQueryStringParams("val="..newValue)
GateHttp->StatePushNotification->SendRequest()
```

Aby poinformować klienta o nowym statusie należy wywołać skrypt podając jako parametr nową wartość cechy Value. Najlepiej zrobić to w zdarzeniu OnValueChange obiektu Relay. Ponieważ przypisaliśmy krok wcześniej wartość do zmiennej użytkownika GateHttp->RelayValueOnGateHttp, możemy jej użyć aby uniknąć niepotrzebnego ponownego przesysłanie tej wartości. Zatem przypisanie wyglądać będzie następująco:

`GateHttp->SendStatePushNotification(GateHttp->RelayValueOnGateHttp)`

Należy pamiętać, że kopiowanie wartości do cechy RelayValueOnGateHttp musi nastąpić wcześniej.

Od teraz za każdym razem gdy wartość cechy Value obiektu Relay się zmieni automatycznie zostanie wysłana notyfikacja z nową wartością.

Wybrana metoda przesyła nową wartość jako parametr URL ale można oczywiście sformatować odpowiedź w dowolny sposób i przesłać w ciele wiadomości ustawiając wartość za pomocą metody SetRequestBody(value).

Podsumowanie

Omówiliśmy podstawowe aspekty integracji systemu Grenton za pomocą Gate'a Http. Omówione zostały typowe problemy, które mogą wystąpić przy takich konfiguracjach oraz metody ich rozwiązania. Stosując się do przedstawionych wytycznych można zrealizować dowolnie złożone wymagania i zaimplementować złożony system tak aby działał stabilnie, niezawodnie i szybko. Należy pamiętać, że Gate Http otwiera nieograniczone możliwości kooperacji z systemem i można za jego pomocą wykonać dowolną operację, nawet szkodliwą. Dlatego ważne jest aby konfiguracja Gate Http była starannie przemyślana i wykonana z najwyższą dbałością.