RPPIOT prakticni materijali

Zlatan Sicanica

stu 29, 2021

Sadržaj

Ovaj dokument sadrzi objasnjenja koda napisanog u repozitorijima s primjerima. Svaki repozitorij odgovara zadatcima/radionicama s predavanja, a ovdje su oni detaljnije objasnjeni, skupa s postupkom rjesavanja.

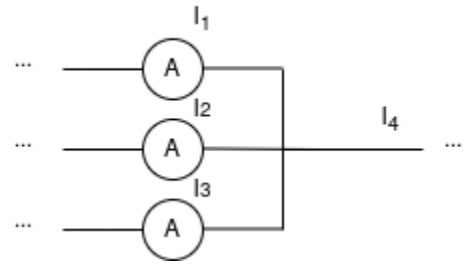
POGLAVLJE 1

Arhitektura

Ovaj primjer oslanja se na drugo predavanje o arhitekturama industrijskih IoT sustava.

1.1 Zadatak

Zamisljena je situacija da imamo sljedeci dio elektricne sheme:



Ampermetri su industrijski uredaji koji svoja stanja notificiraju pomocu IEC104 protokola. Zadatak je citati te podatke, ispisati ih na konzolu i u taj ispis ukljuciti i novu vrijednost, koja bi predstavljala struju I4, zbroj ostale tri jakosti struja.

Implementacija zadatka moze se preuzeti iz repozitorija. Strukturiran je tako da se u paketu `simulator` nalazi implementacija simulatora ampermetara. Simulaciju je potrebno pokrenuti prema uputama iz repozitorija i ispis bi trebao izgledati ovako:

```
(01-architecture) zlatan@quicksilver ~/c/r/01-architecture (master)> python -m simulator.main
changing current 2 from 2 to 2.58
state: [1, 3, 2]
changing current 2 from 2.58 to 4.71
state: [1, 3, 2.58]
changing current 0 from 1 to 3.73
state: [1, 3, 4.71]
changing current 2 from 4.71 to 1.15
state: [3.73, 3, 4.71]
changing current 2 from 1.15 to 3.11
state: [3.73, 3, 1.15]
changing current 2 from 3.11 to 3.2
state: [3.73, 3, 3.11]
changing current 2 from 3.2 to 0.67
state: [3.73, 3, 3.2]
changing current 2 from 0.67 to 1.66
state: [3.73, 3, 0.67]
changing current 1 from 3 to 3.52
state: [3.73, 3, 1.66]
changing current 0 from 3.73 to 4.32
state: [3.73, 3.52, 1.66]
changing current 1 from 3.52 to 1.81
state: [4.32, 3.52, 1.66]
changing current 1 from 1.81 to 0.12
state: [4.32, 1.81, 1.66]
changing current 0 from 4.32 to 0.96
state: [4.32, 0.12, 1.66]
```

U paketu `solution` nalazi se rjesenje zadataka do kojeg smo dosli na predavanju. Ono moze posluziti kao referenca, a u nastavku cemo detaljnije objasniti poduzete korake za dolazak do njega.

1.2 Rjesenje

1.2.1 Priprema

Prvi korak u dolasku do rjesenja je ostvarivanje komunikacije sa simulatorom. Prema njegovim uputama, on svoje podatke posluzuje na adresi `127.0.0.1:9999`, koristeci protokol IEC104. IEC104 je industrijski protokol koji se cesto koristi u podrucju elektroenergetike, a ovdje je iskoristen jer ima relativno jednostavno sucelje.

Ako znamo s kakvim komunikacijskim protokolom uredaj radi, iduci korak za ostvarivanje komunikacije je nalazak odgovarajuce upravljacke biblioteke koja implementira protokol IEC104. Takva biblioteka pruzala bi svojim korisnicima funkcije i klase za otvaranje konekcije s uredajem, slanje i primanje podataka s te konekcije. Jedna implementacija upravljacke biblioteke za IEC104 je biblioteka koju smo koristili na predavanju, hat-drivers iz `hat-open` projekta. Ova biblioteka sadrzi i druge protokole, no u ovom primjeru se samo fokusiramo na IEC104.

1.2.2 asyncio

Gledanjem dokumentacije biblioteke, vidimo da je prva funkcija koju je potrebno koristiti hat.drivers.iec104.connect, koja otvara konekciju s uredajem. Iz potpisa funkcije mozemo vidjeti nove potrebe:

- funkcija je `async`, odnosno, namjenjena je pokretanju kroz Pythonovu `asyncio` infrastrukturu
- adresa na kojoj simulator posluzuje podatke je jedini obvezni argument

Adresa je vec navedena u samom zadatku, a asyncio pokrecemo s ovakvim kodom:

```
import asyncio
import sys


async def async_main():
    pass


def main():
    asyncio.run(async_main())


# standardna dobra praksa za definiranje ulazne tocke
# u Python program
if __name__ == '__main__':
    sys.exit(main())
```

Sad u `async_main` funkciji mozemo koristiti `await` pozive za pokretanje konkurentnih metoda (vise o koristima `asyncio`-a mozete vidjeti u sluzbenoj dokumentaciji). Prakticno, to nam omogucava da koristimo `iec104.connect` funkciju.

1.2.3 „Grubo" rjesenje

U ovom dijelu razviti cemo rjesenje koje prakticno rjesava problem, ali ne uzima u obzir dobre organizacijske prakse kojih se drzimo ako odaberemo neku arhitekturu. Umjesto postojanja tri nezavisne, specijalizirane komponente, cijeli IoT sustav bit ce implementiran u jednoj `while` petlji.

Prvi korak je otvaranje veze na ampermetre, moguce ga je napraviti pozivom `await iec104.connect(iec104.Address('127.0.0.1', 9999))`. Ovdje je adresa predana kroz strukturu podataka specificiranu od strane drivera.

Nakon sto smo se spojili na ampermetar, zelimo ocitavati podatke s njega. `connect` funkcija nam je vratila instancu klase hat.drivers.iec104.Connection. Gledanjem dokumentacije te klase, vidimo da to mozemo pomocu metode hat.drivers.iec104.Connection.receive. Ova metoda ne prima nikakve argumente i vraca listu instanci klase hat.drivers.iec104.Data. Ako dodamo ispis vrijednosti koje su primljene kao argument, nase rjesenje trenutno izgleda ovako:

```
from hat.drivers import iec104
import asyncio
import sys


async def async_main():
    connection = await iec104.connect(
        iec104.Address('127.0.0.1', 9999))
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    while True:
        data = await connection.receive()
        print(data)


def main():
    asyncio.run(async_main())


# standardna dobra praksa za definiranje ulazne tocke
# u Python program
if __name__ == '__main__':
    sys.exit(main())
```

A ispis, ako ga pokrenemo paralelno uz simulator izgleda ovako:

```
(materials) zlatan@quicksilver ~/c/r/materials (master) [0|SIGINT]> python 01-architecture/solution2.py
[Data(value=FloatingValue(value=3.5899999141693115), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
 overflow=False), time=None, asdu_address=0, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=4.539999961853027), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
overflow=False), time=None, asdu_address=2, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=4.210000038146973), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
overflow=False), time=None, asdu_address=0, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=4.860000133514404), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
overflow=False), time=None, asdu_address=1, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=2.3499999046325684), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
 overflow=False), time=None, asdu_address=0, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=1.7400000095367432), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
 overflow=False), time=None, asdu_address=2, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=3.549999952316284), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
overflow=False), time=None, asdu_address=2, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=1.4700000286102295), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
 overflow=False), time=None, asdu_address=0, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=0.5), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False, overflow=False
), time=None, asdu_address=2, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
[Data(value=FloatingValue(value=3.0999999046325684), quality=Quality(invalid=False, not_topical=False, substituted=False, blocked=False,
 overflow=False), time=None, asdu_address=0, io_address=0, cause=<Cause.SPONTANEOUS: 3>, is_test=False)]
```

Vidimo kako ispis izgleda dosta neuredno, razlog tome je cinjenica da paketi IEC104 protokola sadrze dosta dodatnih informacija, poput kvalitete podatka, vremena kad je ocitan, i sl., koje nam trenutno nisu potrebne i mozemo ih ignorirati. Konkretno, citanjem opisa zadatka, vidimo da nam je jedino zanimljivo polje, osim `value` u kojem je zapisana vrijednost ocitanja, `asdu_address` jer njega koristimo kao identifikator ampermetra (razlikovanje I1, I2 i I3). Jos jedan suptilni detalj je cinjenica da u `value` nije zapisan direktno broj, vec, kako IEC104 podrzava slanje razlicitih tipova podataka preko istog sucelja, zapisana je instanca klase hat.drivers.iec104.FloatingValue. To oznacava da je preko protokola primljen realni broj i da mu se moze pristupiti preko varijable `hat.drivers.iec104.FloatingValue.value`. Dakle, do identifikatora ampermetra dolazimo preko `data[0].asdu_address`, a do iznosa ocitanog na ampermetru preko `data[0].value.value`. `[0]` je potreban jer je preko `receive` metode moguce primiti vise od jednog podatka, no zadatak je postavljen tako da se uvijek notificira jedna promjena pa je ovakav hack prihvatljiv.

Dodatni zahtjev je kontinuirano izracunavanje vrijednosti I4, zbroja ostale tri struje, i njegov kontinuirani ispis na konzoli. Radi jednostavnosti provjere, ispisivati cemo stanje sve tri struje uz I4. Uz to, radi urednosti ispisa, dodatno cemo zaokruziti sve vrijednosti na dvije decimale. Stanja cemo cuvati u dictionary-ju gdje su nam kljucevi imena struja, a vrijednosti njihovi iznosi. Mozemo ga izvesti ovako:

```
from hat.drivers import iec104
import asyncio
import sys


async def async_main():
    connection = await iec104.connect(
        iec104.Address('127.0.0.1', 9999))
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    state = {'I1': 0, 'I2': 0, 'I3': 0, 'I4': 0}
    while True:
        data = await connection.receive()
        meter = {0: 'I1', 1: 'I2', 2: 'I3'}[data[0].asdu_address]
        state[meter] = round(data[0].value.value, 2)
        state['I4'] = round(state['I1'] + state['I2'] + state['I3'], 2)
        print(state)


def main():
    asyncio.run(async_main())


# standardna dobra praksa za definiranje ulazne tocke
# u Python program
if __name__ == '__main__':
    sys.exit(main())
```

Ispis ce sada izgledati ovako:

```
(materials) zlatan@quicksilver ~/c/r/materials (master)> python 01-architecture/solution3.py
{'I1': 0, 'I2': 2.05, 'I3': 0, 'I4': 2.05}
{'I1': 1.09, 'I2': 2.05, 'I3': 0, 'I4': 3.14}
{'I1': 1.09, 'I2': 2.05, 'I3': 4.85, 'I4': 7.99}
{'I1': 1.09, 'I2': 2.05, 'I3': 2.45, 'I4': 5.59}
{'I1': 1.09, 'I2': 2.05, 'I3': 3.41, 'I4': 6.55}
{'I1': 2.37, 'I2': 2.05, 'I3': 3.41, 'I4': 7.83}
{'I1': 2.37, 'I2': 2.05, 'I3': 0.33, 'I4': 4.75}
{'I1': 1.44, 'I2': 2.05, 'I3': 0.33, 'I4': 3.82}
{'I1': 0.31, 'I2': 2.05, 'I3': 0.33, 'I4': 2.69}
{'I1': 0.31, 'I2': 3.9, 'I3': 0.33, 'I4': 4.54}
{'I1': 0.31, 'I2': 2.68, 'I3': 0.33, 'I4': 3.32}
{'I1': 0.31, 'I2': 3.32, 'I3': 0.33, 'I4': 3.96}
{'I1': 0.31, 'I2': 0.73, 'I3': 0.33, 'I4': 1.37}
{'I1': 0.31, 'I2': 2.24, 'I3': 0.33, 'I4': 2.88}
{'I1': 0.31, 'I2': 2.24, 'I3': 3.44, 'I4': 5.99}
{'I1': 0.31, 'I2': 2.24, 'I3': 0.57, 'I4': 3.12}
{'I1': 0.31, 'I2': 1.74, 'I3': 0.57, 'I4': 2.62}
{'I1': 0.31, 'I2': 1.74, 'I3': 0.6, 'I4': 2.65}
```

Ovime smo zadovoljili minimalne potrebe zadatka, no ne mozemo tvrditi da je rjesenje dugorocno odrzivo. Ako se potrebe promijene, npr. zelimo komunicirati s drugim protokolom, potrebna su dodatna mjerenja, nove vrste izracuna ili drugaciji nacin vizualizacije, takve promjene je tesko implementirati u ovakvo rjesenje. Zbog toga si mozemo pomoci tako da rjesenje implementiramo pomocu specijaliziranih komponenti.

1.2.4 Rjesenje bazirano na predlozenoj arhitekturi

U predavanju je predstavljena okvirna ideja kako generalno izgledaju arhitekture IIoT sustava. Obicno nastanu tri glavne komponente, jedna za komunikaciju s fizickim uredajima, druga za obradu podataka koji se prime s njih i treca za prezentaciju podataka korisniku. Ovakvu vrstu specijalizacije cemo uvesti i u nase rjesenje.

Klase su najjednostavniji nacin kako mozemo definirati odvojene komponente i implementirati njihovu specijaliziranu logiku. Prva komponenta koja se namece je komponenta za komunikaciju s uredajima. Shodno tome, definiramo klasu `Communication` koja sadrzi logiku za ostvarivanje veze s uredajem, primanje podataka s njega i njihovo daljnje slanje u modul za obradu podataka. Zasad nemamo komponentu za obradu podataka, tako da cemo cijelu logiku obrade i prezentacije podataka kopirati u funkciju koja obavlja primanje podataka. Tako dolazimo do sljedece verzije rjesenja:

```
from hat.drivers import iec104
import asyncio
import sys


class Communication:

    def __init__(self):
        self._connection = None

    async def connect(self):
        self._connection = await iec104.connect(
            iec104.Address('127.0.0.1', 9999))

    async def receive_loop(self):
        state = {'I1': 0, 'I2': 0, 'I3': 0, 'I4': 0}
        while True:
            data = await self._connection.receive()
            meter = {0: 'I1',
                     1: 'I2',
                     2: 'I3'}[data[0].asdu_address]
            state[meter] = round(data[0].value.value, 2)
            state['I4'] = round(state['I1']
                                + state['I2']
                                + state['I3'], 2)
            print(state)


async def async_main():
    communication = Communication()
    await communication.connect()
    await communication.receive_loop()


def main():
    asyncio.run(async_main())


# standardna dobra praksa za definiranje ulazne tocke
# u Python program
if __name__ == '__main__':
    sys.exit(main())
```

Ispis rjesenja izgleda isto kao i kod ranije verzije rjesenja. Razlog tome je jednostavan, logika je ostala ista, samo je cijela petlja prebacena u `Communication` klasu. Sad cemo ju dodatno razbiti tako sto cemo dodati novu komponentu za obradu podataka, implementiranu u klasi `Processing`. Ona ima glavnu metodu `process`, koja sadrzi logiku za izracun struje I4 i ispis podataka (zasad, dok ne dodamo komponentu za vizualizaciju). Jos jedan dodatak je da sad `Communication` komponenta treba imati referencu na `Processing` jer je to najjednostavniji nacin da ju „obavijesti" o tome da je primila novo ocitanje. Alternativa bi bila da komuniciraju na neki drugi nacin, npr. preko asyncio.Queue, zapisa i citanja datoteke, preko socketa, ... Tako dolazimo do nove verzije naseg rjesenja:

```
from hat.drivers import iec104
import asyncio
import sys


class Communication:

    def __init__(self, processing):
        self._connection = None
        self._processing = processing

    async def connect(self):
        self._connection = await iec104.connect(
            iec104.Address('127.0.0.1', 9999))

    async def receive_loop(self):
        while True:
            data = await self._connection.receive()
            self._processing.process(data)


class Processing:

    def __init__(self):
        self._state = {'I1': 0, 'I2': 0, 'I3': 0, 'I4': 0}

    def process(self, iec104_data):
        meter = {0: 'I1',
                 1: 'I2',
                 2: 'I3'}[iec104_data[0].asdu_address]
        self._state[meter] = round(iec104_data[0].value.value, 2)
        self._state['I4'] = round(self._state['I1']
                                  + self._state['I2']
                                  + self._state['I3'], 2)
        print(self._state)


async def async_main():
    processing = Processing()
    communication = Communication(processing)
    await communication.connect()
    await communication.receive_loop()


def main():
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    asyncio.run(async_main())


# standardna dobra praksa za definiranje ulazne tocke
# u Python program
if __name__ == '__main__':
    sys.exit(main())
```

Konacno, ako zelimo imati arhitekturu opisanu u predavanju, fali nam jos i komponenta za vizualizaciju. Trenutno se cijelo stanje aplikacije ispisuje diraktno na konzolu, htjeli bismo tu logiku izdvojiti u zasebnu komponentu i mozda izbaciti neki stiliziraniji ispis od Pythonove pretvorbe dictionary-ja u string. Stvaramo novu klasu, `Visual`, koja ima metodu `render`. Ona prima stanje aplikacije u formatu koji propisuje `Processing` klasa, i ispisuje vrijednosti na konzolu. Zaokruzivanje mozemo takoder prebaciti u ovu klasu, posto je ono u ovom slucaju iskljucivo vizualna prilagodba podatka. Slicno kao i kod povezivanja klasa za komunikaciju i obradu podataka, komponenta za obradu podataka ima referencu na komponentu za vizualizaciju, kako bi joj mogla proslijediti nove verzije svog stanja. Alternative takvom obliku komunikacije iste su kao i kod veze komunikacija-obrada. Tako dolazimo do ove verzije rjesenja:

```
from hat.drivers import iec104
import asyncio
import sys


class Communication:

    def __init__(self, processing):
        self._connection = None
        self._processing = processing

    async def connect(self):
        self._connection = await iec104.connect(
            iec104.Address('127.0.0.1', 9999))

    async def receive_loop(self):
        while True:
            data = await self._connection.receive()
            self._processing.process(data)


class Processing:

    def __init__(self, visual):
        self._state = {'I1': 0, 'I2': 0, 'I3': 0, 'I4': 0}
        self._visual = visual

    def process(self, iec104_data):
        meter = {0: 'I1',
                 1: 'I2',
                 2: 'I3'}[iec104_data[0].asdu_address]
        self._state[meter] = iec104_data[0].value.value
        self._state['I4'] = (self._state['I1']
                             + self._state['I2']
                             + self._state['I3'])
```

(continues on next page)

(nastavak sa prethodne stranice)

```
        self._visual.render(self._state)


class Visual:

    def render(self, state):
        for key, value in state.items():
            print(key, '=', round(value, 2))
        print()


async def async_main():
    visual = Visual()
    processing = Processing(visual)
    communication = Communication(processing)
    await communication.connect()
    await communication.receive_loop()


def main():
    asyncio.run(async_main())


# standardna dobra praksa za definiranje ulazne tocke
# u Python program
if __name__ == '__main__':
    sys.exit(main())
```

Ona sad ima i drugaciji ispis, koji izgleda ovako:

```
(materials) zlatan@quicksilver ~/c/r/materials (master) [0|SIGINT]> python 01-architecture/solutions/6_visual_component.py
I1 = 0
I2 = 0
I3 = 4.67
I4 = 4.67

I1 = 0
I2 = 0.76
I3 = 4.67
I4 = 5.43

I1 = 0
I2 = 0.76
I3 = 4.18
I4 = 4.94

I1 = 0.73
I2 = 0.76
I3 = 4.18
I4 = 5.67

I1 = 4.59
I2 = 0.76
I3 = 4.18
I4 = 9.53

I1 = 1.34
I2 = 0.76
I3 = 4.18
I4 = 6.28

I1 = 1.34
I2 = 0.85
I3 = 4.18
I4 = 6.37
```

Ovime smo implementirali rjesenje problema koje je ujedno i arhitekturalno smisleno, odnosno, moguce ga je prosirivati s dolaskom novih zahtjeva. Logicke cjeline su medusobno odvojene i relativno jednostavno ih mozemo prilagodavati po potrebi. Jedna ocita slabost ovako postavljenog sustava je cinjenica da pojedine komponente moraju biti upoznate s nacinom kako funkcioniraju ostale komponente u sustavu. Tako npr, `Communication` klasa mora znati za `Processing` i mora znati sto mu salje u `process` metodu. Ako se ikad dogodi da nam takvo sucelje vise ne bude dovoljno dobro, potrebno je raditi ekstenzivnije modifikacije (mijenjati i `Processing` i `Communication` komponente). I dodatak novih uredaja/komunikacijskih protokola s kojima se radi i dalje zahtjeva reorganizaciju `Communication` dijela. Zbog svega ovoga, pribjegavamo koristenju gotovih rjesenja koja se brinu o infrastrukturi oko dijelova implementacije koji su specificni za konkretni problem koji se rjesava. Konkretne primjere ovoga vidjet cemo kad krenemo raditi s infrastrukturnim komponentama koje su implementeirane u sklopu `hat-open` projekta.

POGLAVLJE 2

Modbus konzolna aplikacija

Ovo poglavlje pokriva prvi dio radionice s 4. predavanja, spajanje na Modbus uredaj preko konzolne aplikacije i ispis ocitane vrijednosti na izlaz. Rjesenje je objavljeno na repozitoriju s predavanja, a ovaj dio opisuje korake potrebne za dolazak do njega.

2.1 Zadatak

Na adresi 161.53.17.239:8502 preko Modbus TCP protokola posluzuju se podatci o temperaturi s termometra. Specifikacija uredaja je dostupna na ovoj adresi. Potrebno je napraviti konzolnu aplikaciju koja ce se spojiti na ovaj uredaj i ispisivati ocitanja temperature. Nije potrebno pretjerano se zamarati s trazenjem optimalne arhitekture, jer ce se kasnije razvijati rjesenje bazirano na tehnologijama iz `hat-open` projekta, koje pokrivaju te probleme.

2.2 Rjesenje

Kao i prosli put, prvi korak je proucavanje komunikacijskog sucelja uredaja s kojim radimo. Gledanjem *specifikacije <https://download.inveo.com.pl/manual/nano_t_poe/user_manual_en.pdf>*, vidimo da podatcima o temperaturi mozemo pristupiti na vise nacina, no mi se u sklopu zadatka fokusiramo na Modbus, cije sucelje je opisano u poglavlju 7.7. U njemu se nalazi nekoliko tablica na temelju kojih mozemo pristupati podatcima, konfigurirati uredaj i sl. Tablice imaju stupce *Address*, *Name*, *R/W* i *Description*. *Address* nam je najzanimljiviji podatak, njega mozemo intepretirati kao identifikator ocitanja. Po Modbus protokolu, ocitanja se modeliraju kao sekvencijalna memorija i klijenti koji se spajaju na Modbus uredaje, pristupaju podatcima tako da salju zahtjeve za citanje odredene adrese. Sto se tice ostalih stupaca, po *Description* mozemo vidjeti semantiku svake adrese. Vidimo da adresa 4004 ima informaciju o temperaturi pomnozenu s 10. To je adresa kojoj cemo pristupati preko nase konzolne aplikacije.

Drugi korak je nabavljanje komunikacijskog drivera za Modbus. To opet mozemo koristiti hat-drivers paket, ovaj put Modbus implementaciju. Vidimo kako ona ima razne funkcije za kreiranje konekcije, na temelju tipa konekcije koju zelimo otvoriti cemo odabrati jednu od `create_...` funkcija. S obzirom da se uredaj ponasa kao slave, to znaci da ce se nasa konzolna aplikacija ponasati kao master. Dodatno, komuniciramo preko TCP-a, ne preko serial porta, tako da cemo koristiti funkciju hat.drivers.modbus.create_tcp_master. Ova funkcija prima dva obvezna argumenta, `modbus_type` i `address`. `modbus_type` je tip Modbus uredaja s kojim

se radi, to je enumeracija definirana od strane biblioteke *hat.drivers.modbus.ModbusType <https://hat-drivers.hat-open.com/py_api/hat/drivers/modbus/common.html#hat.drivers.modbus.common.ModbusType>*. Radimo s TCP-om, tako da je `ModbusType.TCP` ispravan tip. `address` je struktura podataka koja predstavlja TCP adresu, definirana na hat.drivers.tcp.Address. Tu unosimo IP adresu i port na kojoj termometar posluzuje podatke. Uz sve ovo, vidimo da je `create_tcp_master` funkcija `async` te da ju je potrebno pokretati kroz asyncio infrastrukturu.

Temeljem svega spomenutog, dolazimo do prve verzije rjesenja, gdje se samo spajamo na termometar:

```
from hat.drivers import modbus, tcp
import asyncio
import sys


async def async_main():
    master = await modbus.create_tcp_master(
        modbus.ModbusType.TCP,
        tcp.Address('127.0.0.1', 9999))


def main():
    asyncio.run(async_main())


if __name__ == '__main__':
    sys.exit(main())
```

`create_tcp_master` vraca objekt tipa hat.drivers.modbus.Master. Gledanjem njegove konfiguracije, vidimo da on ima funkciju read. Ona ima obvezne argumente `device_id`, `data_type` i `start_address`.

`device_id` oslanja se na cinjenicu da Modbus protokol moze biti realiziran kao multidrop. To znaci da na jednu konekciju moze biti spojeno vise stvarnih uredaja i argumentima poput `device_id`-a se specificira kojem uredaju se treba proslijediti taj zahtjev. U nasem konkretnom slucaju, nemamo vise uredaja u multidropu, pa je prihvatljiva vrijednost za identifikator 1.

`data_type` referencira tip podatka koje Modbus moze posluzivati. Protokol podrzava tipove poput coil i holding register (nazivi iz povjesnih razloga kad se radilo s fizickim registrima i zavojnicama). Po tablici iz specifikacije uredaja, vidimo da je temperatura zapisana u holidng register-u, tako da koristimo enumeraciju `hat.drivers.modbus.DataType.HOLDING_REGISTER` kao `data_type`.

`start_address` je adresa na kojoj je podatak posluzen. Po tablici u dokumentaciji to se posluzuje na adresi 4004, tako da je to vrijednost. Isprobavanjem rjesenja, videno je da je ova informacija zapravo zapisana na adresi 4003, sto je vjerojatno zbog pocetnog indeksa, dokumentacija krece od 1, dok Modbus driver pretpostavlja start od 0. Dakle, `start_address` je 4003.

Recimo da zelimo kontinuirano slati upite za ocitanjem na uredaj svakih 5 sekundi. Onda bi nam rjesenje izgledalo ovako:

```
from hat.drivers import modbus, tcp
import asyncio
import sys


async def async_main():
    master = await modbus.create_tcp_master(
        modbus.ModbusType.TCP,
        tcp.Address('161.53.17.239', 8502))
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    while True:
        data = await master.read(
            device_id=1,
            data_type=modbus.DataType.HOLDING_REGISTER,
            start_address=4003)
        print(data)
        await asyncio.sleep(5)


def main():
    asyncio.run(async_main())


if __name__ == '__main__':
    sys.exit(main())
```

Ovime na konzolni ispis dobivamo temperaturu pomnozenu s 10. Rjesenje bi se dalo raspisivati detaljnije, kao u prvom zadatku, uvoditi konkretnu arhitekturu i sl., no ovdje je ideja nastaviti s Hat tehnologijama. Iduci zadatak opisuje kako izvesti istu stvar, koristenjem Hatove infrastrukture.

POGLAVLJE 3

Hat

Prije nastavka rjesavanja problema s termometrom, dati cemo uvod u glavne aspekte komponenti Hat projekta, jer cemo se oslanjati na njih za rjesenje. Ovo poglavlje je zamisljeno kao prirucnik tim komponentama, i mozete mu se vratiti kad god zapnete.

U drugom predavanju upoznali smo se s glavnim industrijskim praksama vezanim uz organizaciju koda, i ovdje cemo ih dosta referencirati. Vidjeli smo kako industrijski IoT sustavi obicno imaju ovakvu generalnu arhitekturu:

Takoder smo spominjali koristi izvodenja takve arhitekture pomocu event-driven sustavi. Implementacija takvih sustava moze se vizualizirati sljedecim dijagramom:

Vidimo kako postoje razliciti aktori u sustavu izmedu kojih se nalazi *Event bus*, sabirnica dogadaja. Ideja je da kreatori notificiraju sabirnicu dogadaja o novonastalim promjenama, a onda sabirnica proslijedi te informacije svim zainteresiranim konzumentima. Dogadaj se obicno modelira kao neka struktura koja sadrzi informaciju o semantickom znacenju dogadaja (npr. detektirana je promjena temperature) i konkretne podatke specificne za promjenu (npr. iznos temperature), a mogu biti i popraceni razlicitim vremenskim oznakama (kad je napravljeno ocitanje), dodatnim zastavicama itd.

U kontekstu industrijskih IoT sustava, generalna arhitektura predstavljena ranije, moze se izvesti kroz event-driven sustav. Jedan nacin kako bi ona mogla biti izvedena je sljedeci:

Vidimo da su glavne komponente u principu ostale iste, glavna razlika je da su one sad povezane preko sabirnice dogadaja, dok prije nije bilo egzaktno specificirano kako komuniciraju. Komponente iz hat-open projekta imaju ovakvu arhitekturu. Glavne komponente kojima cemo se baviti u nasim zadatcima su event server (event bus + business logic), gateway (communication) i GUI server (human-machine interface). Uz njih, projekt iz kojeg cemo razvijati nase rjesenje konfigurira i druge komponente, ali njima cemo pristupati samo iz prespektive korisnika, a manje raditi neki konkretan razvoj vezan uz njih.

3.1 hat-event (event bus)

Prva komponenta koju gledamo je event server. Ona zapravo ima dvostranu ulogu, prva je da se ponasa kao sabirnica dogadaja, a druga da sadrzi specijalizirane module za poslovnu logiku. U ovom dijelu fokusiramo se na prvi aspekt, komunikacijsku sabirnicu. Trenutno cemo napraviti malu digresiju od primjera s termometrom, da pokazemo neke generalne ideje oko rada s event serverom, koje su primjenjive u cijelom sustavu.

Event server se pokrece pozivom naredbe `hat-event`. Ta naredba prima komandnolinijski argument `--conf` kojim joj se predaje putanja do konfiguracijske datoteke. Ta datoteka je u JSON ili YAML formatu i ima strukturu propisanu JSON shemom. Minimalna konfiguracija mogla bi biti:

```
---
backend_engine:
    backend:
        module: hat.event.server.backends.dummy
    server_id: 1
communication:
    address: tcp+sbs://127.0.0.1:23012
log:
    version: 1
module_engine:
    modules: []
type: event
...
```

Pozivom `hat-event --conf conf.yaml` (ako je minimalna konfiguracija zapisana u datoteci `conf.yaml`) trebao bi se pokrenuti program bez ikakvog ispisa koji ne zavrsava. Ovaj poziv pokrece sabirnicu dogadaja koja ceka da se na nju spoje aktori event-driven sustava (proizvodaci i potrosaci dogdaja). Iduci korak je implementacija aktora. Spajanje na event server radi se preko hat.event.client modula. Taj modul u sebi sadrzi implementaciju funkcija za spajanje na event server, primanje i registraciju dogadaja. Funkcija `connect` obavlja spajanje na server i vraca nazad instancu klase hat.event.client.Client. Pozivanjem metoda `receive` i `register` se primaju ili registriraju dogadaji.

Dogadaji su uredene trojke koje sadrze atribute `event_type`, `payload` i `source_timestamp`, a konkretna struktura koja se koristi ovisi o tome koja metoda se pokusava zvati (npr. metoda `receive` vraca hat.event.common.Event, dok se metodi `register` predaje hat.event.common.RegisterEvent, ali obje imaju parametre `event_type`, `payload` i `source_timestamp`, razlika je u tome da `hat.event.common.Event` ima neke dodatne parametre koje mu dodijeli server). `event_type` tuple stringova koja sadrzi semanticko znacenje promjene koja se desila. Po ranije primjeru, `event_type` za dogadaj koji signalizira promjenu ocitanja mjerenja mogao bi biti `['thermometer1', 'measurement_change']`. `payload` sadrzi podatke specificne za promjenu koja se desila, npr. za primjer promjene mjerenja, on bi mogao biti broj koji ozacava novoizmjerenu temperaturu. `source_timestamp` je opcionalna vremenska oznaka u kojoj kreator dogadaja tom dogadaju moze pridruziti oznaku vremena (npr. kad je izmjerena temperatura). Kako je `source_timestamp` opcionalan, u svim primjerima cemo ga stavljati u `None`.

3.1.1 Kreator dogadaja

S ovime na umu, mozemo implementirati prvu skriptu koja ce se ponasti kao kreator dogadaja. Ona se pokrece, spaja na event server i registrira dogadaj koji signalizira promjenu nekog arbitrarnog mjerenja. Funkcijom `connect` spojiti cemo se na event server. U konfiguraciji servera, pod `communication/address` vidimo adresu i port na kojoj server slusa. Tu adresu predajemo prvom argumentu `connect` funkcije. Drugi argument, `subscriptions` zasad cemo ostaviti kao praznu listu, a objasnit cemo ga kad cemo implementirati konzumenta dogadaja za ovaj primjer.

Nakon toga, u beskonacnoj petlji, svake tri sekunde registriramo event ciji je `event_type ('measurement1', 'change', 'abc')`, `source_timestamp` je `None`, a `payload` je JSON-serijalizabilna struktura podataka s jednim

atributom, `value`, cija vrijednost je nasumicni broj od 0 do 10. To nas ostavlja s ovakvom konkretnom implementacijom:

```
import hat.event.client
import hat.event.common
import asyncio
import random
import sys


async def async_main():
    client = await hat.event.client.connect(
        'tcp+sbs://127.0.0.1:23012', [])

    while True:
        client.register([hat.event.common.RegisterEvent(
            event_type=('measurement1', 'change', 'abc'),
            source_timestamp=None,
            payload=hat.event.common.EventPayload(
                type=hat.event.common.EventPayloadType.JSON,
                data={'value': random.randint(0, 10)}))])
        await asyncio.sleep(3)


def main():
    asyncio.run(async_main())


if __name__ == '__main__':
    sys.exit(main())
```

Ovaj program ce se upaliti i raditi dok ga se ne ugasi, bez ispisivanja icega, ali mozete dodati ispise nasumicnih brojeva koji se registriraju. Takoder, umjesto `register`, postoji metoda register_with_response koja vrati nazad instance dogadaja koji su se registrirali, zanimljivo bi bilo vidjeti njihov ispis.

3.1.2 Konzument dogadaja

Mogucnost registracije dogadaja nam nema puno koristi ako se ti dogadaji ne propagiraju do nekih drugih klijenata. Zbog toga imamo potrebu razviti novog aktora koji bi primao dogadaje koje registrira kreator iz proslog dijela, i ispisivao ih na konzolu. U proslom dijelu smo kod `connect` funkcije ignorirali argument `subscriptions` jer nam tad nije bio potreban, sad cemo ga koristiti da novostvorenog klijenta „pretplatimo" na dogadaje s odredenom semantikom. Semantiku dogadaja odreduje njegov `event_type`, koji je izveden kao tuple stringova. Ako pogledamo potpis connect funkcije, vidimo da je pretplata definirana kao lista tuplova stringova, odnosno novostvoreni klijent se pretplacuje na n tipova dogadaja.

Nakon stvaranja konekcije s pretplatom, iduca metoda klijenta koja nas zanima je receive. Kad event server primi dogadaj s tipom koji na koji je klijent pretplacen, on mu ga posalje. `receive` metoda ceka da klijent primi dogadaj i vrati ga na izlaz.

Na temelju ovoga, mozemo implementirati naseg konzumenta:

```
import hat.event.client
import hat.event.common
import asyncio
```

(continues on next page)

(nastavak sa prethodne stranice)

```
import random
import sys


async def async_main():
    client = await hat.event.client.connect(
        'tcp+sbs://127.0.0.1:23012', [
            ('measurement1', 'change', 'abc')])

    while True:
        events = await client.receive()
        print(events)


def main():
    asyncio.run(async_main())


if __name__ == '__main__':
    sys.exit(main())
```

Pokrenemo li konzumenta i kreatora istovremeno, vidjet cemo da konzument ispisuje dogadaje koje kreator registrira. Pokrenemo li vise kreatora i konzumenata istovremeno, svaki konzument ce ispisivati dogdaje koje registriraju svi kreatori.

Dodatno, kod implementacije konzumenta, pretplatili smo ga na dogadaje s tipom `('measurement1', 'change', 'abc')`, ali implementacija klijenta nam omogucuje i koristenje *wildcard* elemenata `'*'` i `'?'`. `?` moze biti na bilo kojem mjestu unutar pretplate i daje do znanja event serveru da nam je svejedno sto se nalazi u tipu dogadaja na tom mjestu. Tako bi legitimna pretplata bila `('measurement', '?', 'abc')` i konzument bi funkcionirao jednako. Razlika je da, ako bismo imali aktora koji registrira dogadaje s tipom `('measurement', 'xyz', 'abc')`, i ti dogadaji bi se ispisivali. Wildcard `'*'` moze biti samo na kraju pretplate i daje do znanja event serveru da nam je svejedno sto se nalazi u tipu dogadaja od mjesta gdje je znak postavljen. Tako bi pretplata `('measurement', '*')` pokrivala `('measurement', 'change', 'abc')` dogadaje, ali i `('measurement')`, `('measurement', 'xyz')`, `('measurement', '123', '456')`...

3.1.3 Upiti u stare dogadaje

Ovaj aspekt event servera nam mozda nece biti potreban u prakticnim zadatcima, no svejedno ga navodimo radi kompletnosti. Klijenti event servera imaju jos jednu metodu koju nismo pokrili, query. Vidimo po potpisu funkcije da ona predaje argument tipa QueryData. Gledanjem dokumentacije te strukture, vidimo da ona ima puno opcionalnih argumenata koji izgledaju kao filteri. Kad nad klijentom pozovemo `query`, event server primi filtere i na temelju njih napravi upit u bazu podataka kojim pristupi starim dogadajima koji su se registrirali. Onda vrati te dogadaje i klijent ih izbaci kao rezultat poziva metode `query`.

QueryData ima razne argumente:

- `event_ids` filtrira samo one dogadaje s identifikatorima koji su zadani
- `event_types` filtrira dogadaje na temelju njihovog tipa (takoder moze imati wildcardove)
- `t_from`, `t_to` odreduju pocetak i kraj vremenskog intervala `timestamp` parametra dogadaja (serverova vremenska oznaka)
- `source_t_from`, `source_t_to` odreduju pocetak i kraj vremenskog intervala `source_timestamp` parametra dogadaja (klijentova vremenska oznaka)

- `payload` filtrira na temelju `payload` parametra dogadaja, gleda se jednakost
- `order_by` odreduje kako ce vraceni dogadaji biti soritrani
- `unique_type` daje do znanja event serveru da u rezultatu upita ne vrati vise dogadaja s istim tipom
- `max_results` je cvrsto ogranicenje na maksimalni broj dogdaja koji su vraceni

Mozemo i isprobati ovu funkcionalnost, dosadasnja konfiguracija koristi implementaciju baze koja ne radi nista, tako da ju je potrebno malo prilagoditi:

```
---
backend_engine:
    backend:
        module: hat.event.server.backends.sqlite
        db_path: hat-event.db
        query_pool_size: 1
    server_id: 1
communication:
    address: tcp+sbs://127.0.0.1:23012
log:
    version: 1
module_engine:
    modules: []
type: event
...
```

Nakon toga, mozemo pokrenuti kreatora dogadaja, da nam registrira `('measurement', 'change', 'abc')` dogadaje. Paralelno mozemo pokrenuti sljedecu skriptu koja radi upit na bazu:

```
import hat.event.client
import hat.event.common
import asyncio
import random
import sys


async def async_main():
    client = await hat.event.client.connect(
        'tcp+sbs://127.0.0.1:23012', [])

    events = await client.query(
        hat.event.common.QueryData(
            event_types=[
                ('measurement1', 'change', 'abc')]))
    print(events)


def main():
    asyncio.run(async_main())


if __name__ == '__main__':
    sys.exit(main())
```

Na konzoli bi se trebali ispisati svi dogadaji koje je kreator registrirao u proslosti.

Ovime smo pokrili osnove rada s event serverom koji je zajednicki kod svih komponenti, a u nastavku cemo vidjeti kako Hat komponente i njihovi specijalizirani moduli koriste tu infrastrukturu da medusobno suraduju i implementiraju funkcionalnost industrijskih IoT sustava.

3.2 Komponente

Podsjetimo se jos jednom na event-driven arhitekturu industrijskog IoT sustava:

Spomenuli smo da komponente Hat projekta preslikavaju ovu arhitekturu, na sljedeci nacin:

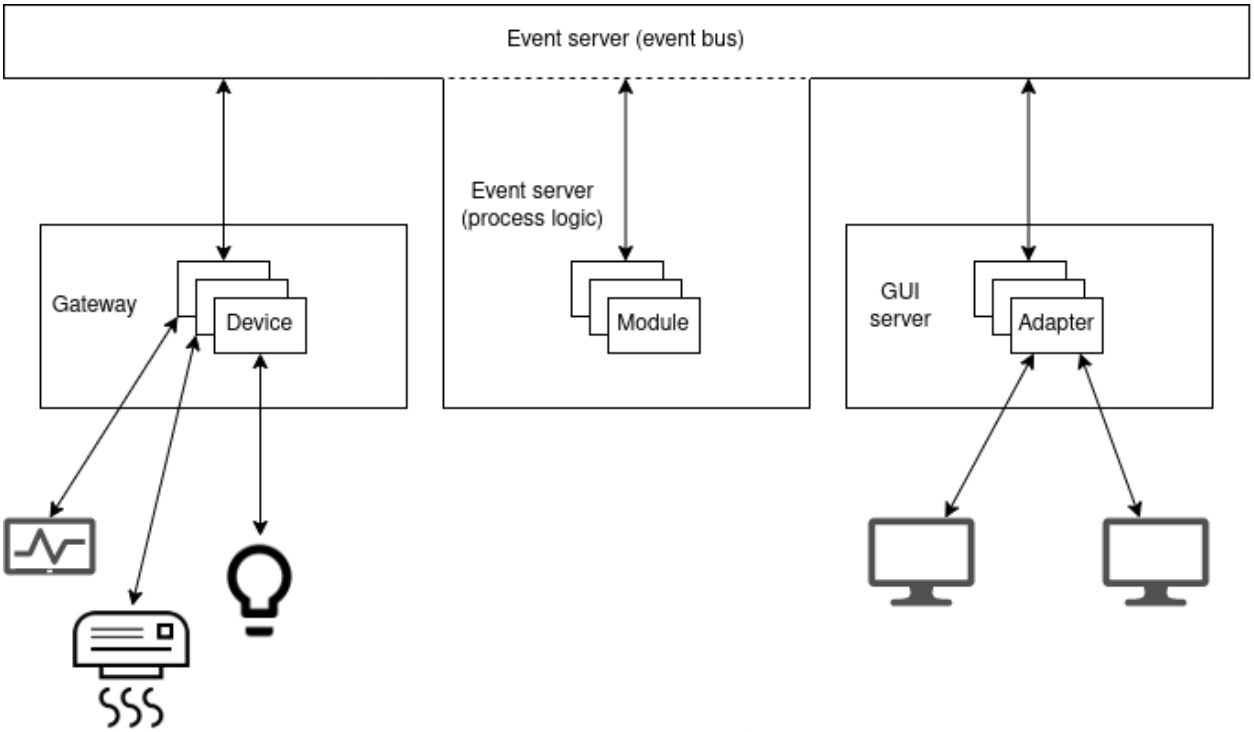
- hat-gateway komponenta je zaduzena za komunikaciju s uredajima
- specijalizirani moduli hat-event komponente su zaduzeni za implementaciju procesne logike
- hat-gui komponenta je zaduzena za vizualizaciju

Sve tri komponente imaju jednu zajednicku crtu, a to je da su same implementacije komponenti genericne, ali se konfiguriraju da koriste implementacije specijaliziranih modula u kojima je sadrzana konkretna domenska logika specificna za aplikaciju koja se razvija. To konkretno znaci da komponenta specificira odredeno sucelje koje modul mora zadovoljiti, implementator sustava napravi implementaciju tog sucelja i kad pokrece komponentu, u konfiguraciji joj zada da koristi tu implementaciju. Ovako se onda komponenta vise brine za „infrastrukturne" stvari, poput spajanja na event server, suradivanja sa specijaliziranim modulima, itd., a specijalizirani moduli implementiraju aplikaciju.

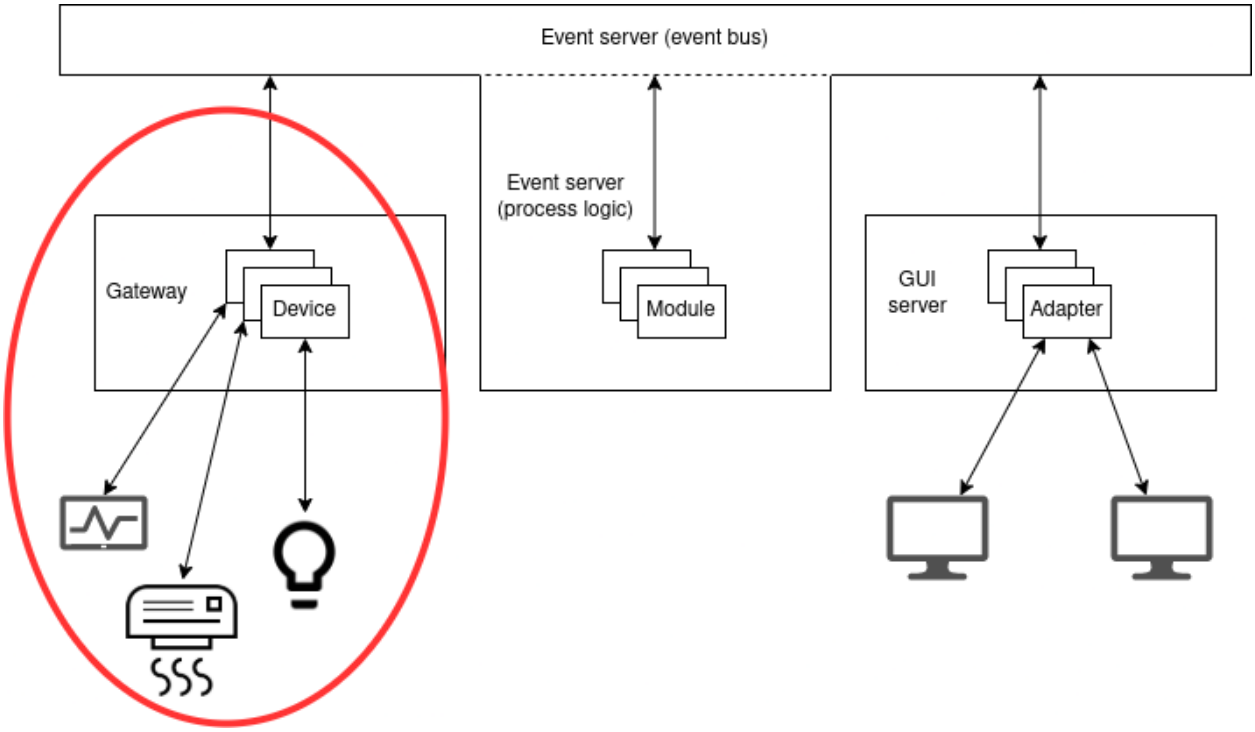
Prije nego sto se bacimo na konkretne komponente, kratki pregled terminologije koju cemo koristiti od sad:

- device - specijalizirani modul gateway komponente
- modul - specijalizirani modul event servera
- adapter - specijalizirani modul GUI komponente

Pogledajmo sad arhitekturu sustava, sad kad znamo za specijalizirane module:



3.2.1 Gateway



Gateway komponenta upravlja svojim specijaliziranim modulima, devicevima, cija zaduzenja su komunikacija s uredajima i pretvorba podataka koje prime preko te komunikacije u dogadaje. Ona se pokrece pozivom `hat-gateway` kojem se preko argument `--conf` zadaje konfiguracija u JSON ili YAML formatu. Konfiguracija je specificirana JSON shemom, a jedan minimalni primjer mogao bi biti:

```
---
type: gateway
event_server_address: tcp+sbs://127.0.0.1:23012
gateway_name: gateway1
devices:
    - module: devices.ammeter
      name: ammeter1
log:
    disable_existing_loggers: false
    formatters:
        default: {}
    handlers:
        console:
            class: logging.StreamHandler
            level: INFO
            stream: ext://sys.stdout
    root:
        handlers:
        - console
        level: INFO
    version: 1
...
```

Najzanimljiviji argumenti ovdje su nam `devices` i `gateway_name`. `gateway_name` ce nam biti bitan kasnije jer ce biti sadrzan u tipu dogadaja s kojim rade devicevi (i njihovi konzumenti). `devices` sadrzi postavke specijaliziranih modula, konfigurira se jedan device ciji modul je implementiran u `devices.ammeter` (gateway ce u nekom trenutku pozvati liniju `import devices.ammeter`). On ima pridruzen i `name` koji ima slicnu svrhu kao i `gateway_name`, bit ce bitan kasnije jer ce biti sadrzan u tipu dogadaja konkretnog devicea. `log` polje mozda izgleda zastrasujuce, ali to je zapravo samo konfiguracija Pythonvog `logging` modula koja se ovdje konfigurira da ispisuje logove na konzolu.

Pogledajmo sada sucelje koje pojedina implementacija devicea mora zadovoljiti. Vidimo da ona mora biti izvedena kao Python modul, koji ima globalne varijable `device_type`, `json_schema_id` i `json_schema_repo`, te funkciju `create`. `device_type` sluzi za klasifikaciju tipa uredaja s kojim komuniciramo, obicno bude jednak imenu protokola koji se koristi. Namjena mu je slicna kao i ranije spomenutim `gateway_name` i `device_name` konfiguracijskim parametrima, budu elementi unutar tipa dogadaja koji se odnose na taj device. `json_schema_id` i `json_schema_repo` su opcionalni pa ih necemo koristiti, a odnose se na mogucnost konfiguriranja devicea. Svaki device moze propisivati svoju strukturu konfiguracije, a format za specifikaciju te strukture je JSON shema. Repo sadrzi shemu a ID kaze s kojim ID-em u shemi se konfiguracija usporeduje. Opcionalni su, tako da ce u nasim primjerima uvijek biti `None`.

`create` funkcija zaduzena je za stvaranje instance klase Device. Po dokumentaciji, ona prima tri argumenta: event klijent, konfiguraciju i „prefiks" tipa dogadaja. S event klijentom smo se bavili u proslom dijelu, takvog klijenta primamo ovdje i mozemo ga koristiti na isti nacin. Moze se primjetiti da on nije bas istog tipa kao i event klijent iz proslog dijela, razlog tome je cinjenica da gateway komponenta stvara svoj wrapper oko originalne instance iz nekih infrastrukturnih razloga (jedan „pravi" klijent za cijeli gateway, odredba pretplata itd.). Konfiguracija je drugi argument, ona je jednaka bilo cemu sto se zapise u konfiguraciji gatewaya u elementima polja `devices`. Treci argument, prefiks tipa dogadaja je zapravo tuple stringova `('gateway', gateway_name, device_type, device_name)`. Za tip dogadaja smo odredili da je definiran kao tuple stringova, svi dogadaji s kojima device radi moraju imati ovaj prefiks. To znaci da svaki dogadaj kojeg device registrira bi trebao pocinjati s ova 4 stringa, npr. `('gateway', 'gateway1', 'iec104', 'iec104_device1', 'measurement_change')`, ako je ime gatewaya `gateway1`, tip devicea `iec104`, a ime `iec104_device1`. Gatway komponenta ne prisiljava da bude ovaj prefiks, to je samo dobra praksa. Ista stvar vrijedi i za primanje dogadaja, `receive` metoda event klijenta kojeg device primi u `create` funkciji vracati ce samo dogadaje kao da je pretplacena na `(*event_type_prefix, '*')`. Nije specificirano u prefiksu, ali nakon njega se obicno navodi smjer komunikacije, odnosno ako dogadaj registrira device onda je to `gateway` a ako ga registrira neki drugi aktor a device ga treba primati, smjer je `system`. Opet, komponenta ne prisiljava ovo ali se potice.

Sad mozemo pogledati neku konkretnu implementaciju gateway devicea. Uzeti cemo raniji primjer s ampermetrima i pretvoriti klasu funkcije `Communication` i preraditi ju da vise ne zove `process`, vec samo registrira dogadaj i ne brine sto se dalje dogada s njim:

```
from hat.drivers import iec104
import asyncio
import hat.aio
import hat.event.common
import hat.gateway.common


json_schema_id = None
json_schema_repo = None
device_type = 'ammeter'


async def create(conf, event_client, event_type_prefix):
    device = AmmeterDevice()

    device._async_group = hat.aio.Group()
    device._event_client = event_client
    device._event_type_prefix = event_type_prefix
    device._async_group.spawn(device._main_loop)
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    return device


class AmmeterDevice(hat.gateway.common.Device):

    @property
    def async_group(self):
        return self._async_group

    async def _main_loop(self):
        connection = await iec104.connect(
            iec104.Address('127.0.0.1', 9999))
        while True:
            data = (await connection.receive())[0]
            self._event_client.register([
                hat.event.common.RegisterEvent(
                    event_type=(*self._event_type_prefix,
                                'gateway', str(data.asdu_address)),
                    source_timestamp=None,
                    payload=hat.event.common.EventPayload(
                        type=hat.event.common.EventPayloadType.JSON,
                        data=data.value.value))])
```

Jedna nejasnoca koja bi se mogla javiti citanjem ovog koda je svrha `hat.aio.Group` klase, koristenja njene `spawn` metode, njenog vracanja kroz property `async_group`, ... Property `async_group` je potreban zbog sucelja koje propisuje `hat.gateway.common.Device` (on nasljeduje `hat.aio.Resource`, a on propisuje da mora postojati taj property). Ideja je da se instanca tog objekta koristi za odredivanje zivotnog ciklusa devicea. Instanca moze biti u otvorenom ili zatvorenom stanju, otvoreno stanje oznacava da device treba raditi, a zatvoreno da ne treba. Metoda `spawn` od grupe ponasa se slicno kao asyncio.create_task, glavna razlika je da Task koji bi se vratio se veze uz stanje otvorenosti grupe - ako se grupa ikad zatvori, Task ce se otkazati (poziv `Task.cancel`) - ovo nam ide u korist jer onda stanje otvorenosti grupe zaista upravlja cinjenicom izvrsava li se `_main_loop` ili ne.

Jedan detalj nismo spomenuli, a bitan je za pokretanje gatewaya s deviceom, je potreba za registracijom dogadaja za paljenje devicea. U dokumentaciji mozemo vidjeti kakvu strukturu imaju ti dogadaji. Dakle, potrebno je registrirati dogadaj s tipom `(*prefiks, 'system', 'enable')` i payloadom `True` jer to signalizira gateway komponenti da pokrene device koji smo konfigurirali. Najjednostavnije to mozemo napraviti s odvojenom skriptom:

```
import hat.event.client
import hat.event.common
import asyncio
import random
import sys


async def async_main():
    client = await hat.event.client.connect(
        'tcp+sbs://127.0.0.1:23012', [])

    await client.register_with_response([
        hat.event.common.RegisterEvent(
            event_type=('gateway', 'gateway1', 'ammeter',
                        'ammeter1', 'system', 'enable'),
```

(continues on next page)

(nastavak sa prethodne stranice)

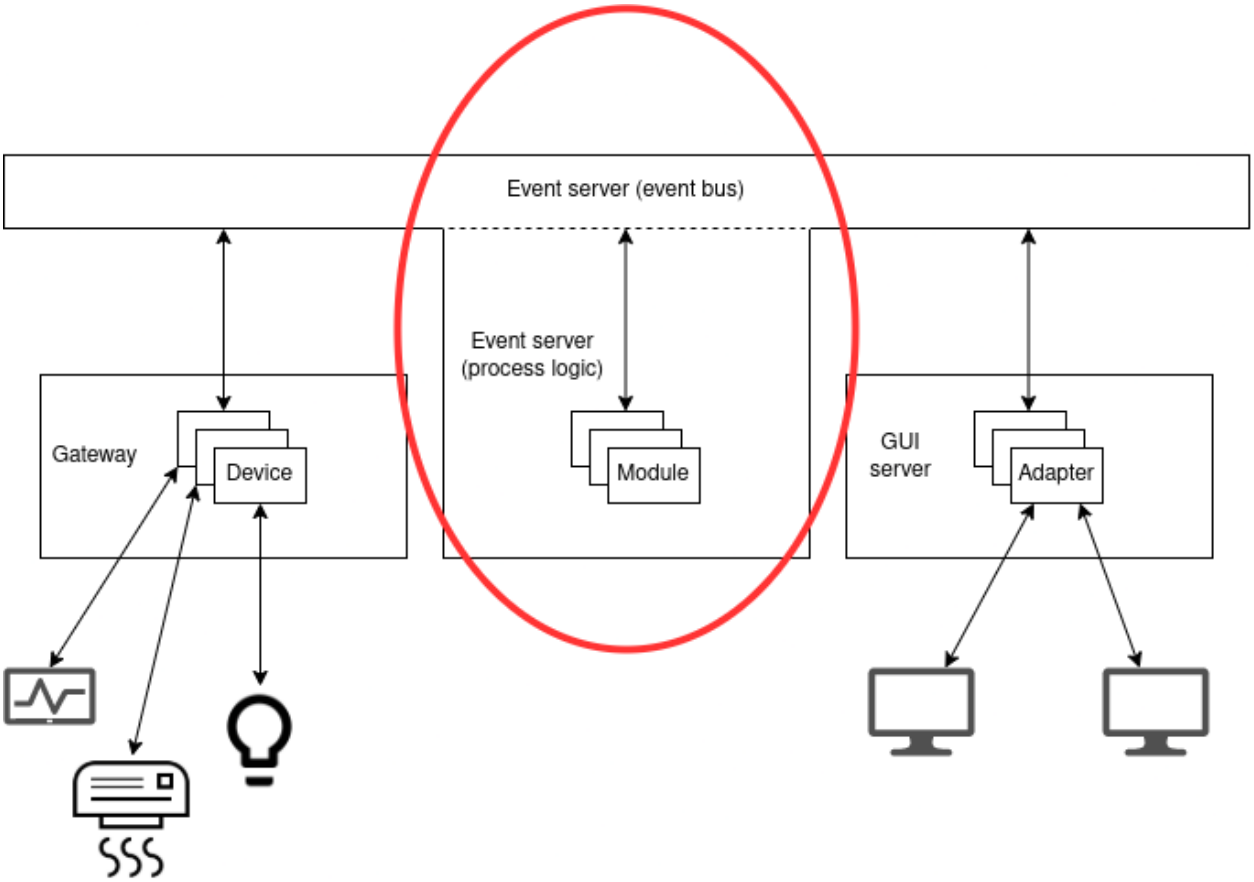
```
            source_timestamp=None,
            payload=hat.event.common.EventPayload(
                type=hat.event.common.EventPayloadType.JSON,
                data=True))])


def main():
    asyncio.run(async_main())


if __name__ == '__main__':
    sys.exit(main())
```

Jos detalja za pokretanje, event server mora biti startan, gateway ga ocekuje na adresi `tcp+sbs://127.0.0.1:23012`. Uz to, treba pokrenuti i simulator iz prvog zadatka, inace ce device izbaciti exception jer connect nece proci.

Prilagodimo li sad naseg ranijeg konzumenta dogadaja (ili skriptu za upite) da prate dogadaje tipa `('gateway', 'gateway1', 'ammeter', 'ammeter1', '*')`, vidjeli bismo da se zaista registriraju ovi dogadaji, nakon `ammeter` dijela je identifikator mjerenja (ASDU adresa iz proslog zadatka), a payload je broj koji predstavlja iznos mjerenja. Ako ima problema oko starta gatewaya, tipa dolazi do ispisa greske `No module named ...`, dodajte direktorij iz kojeg pokrecete u environment varijablu `PYTHONPATH` (linux (bash), windows).

Iduci korak je procesna logika u kojem cemo razviti specijalizirani modul koji ce primati informacije o strujama od devicea i stvarati nove dogadaje na temelju njih.

3.2.2 Event server



Event server smo vec vidjeli u situacijama gdje implementira sabirnicu dogadaja, a sad cemo vidjeti kako pomocu njegovih specijaliziranih modula mozemo implementirati procesnu logiku aplikacije. Ona se izvodi tako da se specijalizirani moduli event servera pretplate na dogadaje odredenog tipa i, kad se dogadaji s tim tipom registriraju, stvore nove dogadaje na temelju njih.

Mozemo preuzeti konfiguraciju iz proslog dijela, glavna razlika je da cemo sad u `module_engine/modules` dodati konfiguraciju specijaliziranog modula:

```
---
type: event
backend_engine:
    backend:
        module: hat.event.server.backends.dummy
    server_id: 1
communication:
    address: tcp+sbs://127.0.0.1:23012
module_engine:
    modules:
      - module: modules.state
log:
    disable_existing_loggers: false
    formatters:
        default: {}
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    handlers:
        console:
            class: logging.StreamHandler
            level: INFO
            stream: ext://sys.stdout
    root:
        handlers:
        - console
        level: INFO
    version: 1
...
```

Dakle, u odnosu na dio gdje smo se fokusirali na nacin kako slati dogadaje preko event servera, ovdje je razlika da smo dodali specijalizirani modul `modules.state` (kao i kod gatewaya, ovo je Python ime modula, u nekom trenutku event server ce zvati `import modules.state`).

Sad je potrebno zaista implementirati modul. Gledanjem dokumentacije, mozemo vidjeti da se to radi tako da definiramo modul tako da implementiramo Python modul koji ima globalne varijable `json_schema_id`, `json_schema_repo` i funkciju `create`. Kod globalnih varijabli vrijedi ista prica kao i kod deviceva, a `create` je korutina koja vraca instancu klase `hat.event.server.common.Module`. Ona prima konfiguraciju modula i referencu na instancu klase `hat.event.module_engine.ModuleEngine`. Konfiguracija modula je iz konfiguracije cijelog event servera, ono sto je napisano uz `module: <Python ime modula>`, a module engine je objekt koji sluzi kao sucelje event servera prema modulu. Ako pogledamo njegovu dokumentaciju, vidimo da ima slicne metode kao event klijent.

Kao i device, modul nasljeduje `hat.aio.Resource` abstraktnu klasu, pa mora imati `async_group` property. Uz njega, ima i property `subscription` kojim se specificira na kakve dogadaje se modul pretplacuje (mala razlika je da to sad vise nije lista tupleova, vec se predaje `hat.event.common.subscription.Subscription` objektu).

Metoda `create_session` je iduca komplikacija. Ideja je da moduli zapravo ne obavljaju registraciju dogadaja sami po sebi, vec da stvaraju sesije koje to rade za njih. Ovo je vise do implementacijskih detalja event servera, gdje kad se registrira dogadaj, event server stvori sesiju svakog modula i onda, ako se modul pretplacuje na dogadaj koji se registrirao, koristi tu sesiju da stvori nove dogadaje. Implementator modula ima korist od toga jer moze imati distinkciju izmedu razlicitih sekvenci obrada podataka, necega za cime nemamo potrebu u sklopu nasih zadataka - zbog toga cemo obradu podataka u sesiji obicno samo proslijediti nazad modulu. `create_session` ne prima nikakve argumente, a vraca instancu objekta `hat.event.server.common.ModuleSession`.

`ModuleSession` je abstraktna klasa koja nasljeduje `hat.aio.Resource`, dakle ima property `async_group` iz istih razloga kao i device i modul. Uz to, ima i metodu `process` koja prima i vraca listu dogadaja. Lista koju prima je sadrzi dogadaje na koje se modul predplacuje kroz `subscription` property, a lista koju vraca je sadrzi nove dogadaje koje zeli registrirati. Mala razlika u odnosu na dosadasnji rad s registracijom dogdaja je da se ovdje ne koristi `RegisterEvent`, vec je potrebno vratiti `hat.event.server.common.ProcessEvent`. On se stvara pozivom module engineove metode `create_process_event`. On prima dogadaj i identifikator izvora dogadaja, hat.event.server.common.Source, pa je dodatno u sesiji potrebno negdje drzati referencu i na njega.

Uz sve ovo imamo dovoljno informacija da napravimo primjer event server modula. Uzeti cemo opet slucaj iz prvog zadatka s ampermetrima, a ovdje cemo napraviti modul koji ce raditi istu stvar kao i `Processing` klasa, uparivanje primljenog mjerenja sa strujama I1, I2 i I3, te racunanje struje I4. Ona bi izgledala ovako:

```
import hat.aio
import hat.event.server.common


json_schema_id = None
json_schema_repo = None

```

(continues on next page)

(nastavak sa prethodne stranice)

```
async def create(conf, engine):
    module = StateModule()

    global _source_id
    module._source = hat.event.server.common.Source(
        type=hat.event.server.common.SourceType.MODULE,
        name='modules.state',
        id=1)

    module._subscription = hat.event.server.common.Subscription([
        ('gateway', 'gateway1', 'ammeter', 'ammeter1', 'gateway', '?')])
    module._async_group = hat.aio.Group()
    module._engine = engine
    module._state = {'I1': 0, 'I2': 0, 'I3': 0, 'I4': 0}

    return module


class StateModule(hat.event.server.common.Module):

    @property
    def async_group(self):
        return self._async_group

    @property
    def subscription(self):
        return self._subscription

    async def create_session(self):
        return StateModuleSession(self, self._async_group.create_subgroup())

    def module_process(self, changes):
        event = changes[0]
        # dohvat zadnjeg elementa tipa dogadaja, za uparivanje s I1, I2, I3
        measurement_id = event.event_type[-1]
        current = {'0': 'I1',
                   '1': 'I2',
                   '2': 'I3'}.get(measurement_id)
        if current is None:
            return []
        self._state[current] = event.payload.data
        self._state['I4'] = (self._state['I1']
                             + self._state['I2']
                             + self._state['I3'])
        return [
            self._engine.create_process_event(
                self._source,
                hat.event.server.common.RegisterEvent(
                    event_type=('state', ),
                    source_timestamp=None,
                    payload=hat.event.server.common.EventPayload(
```

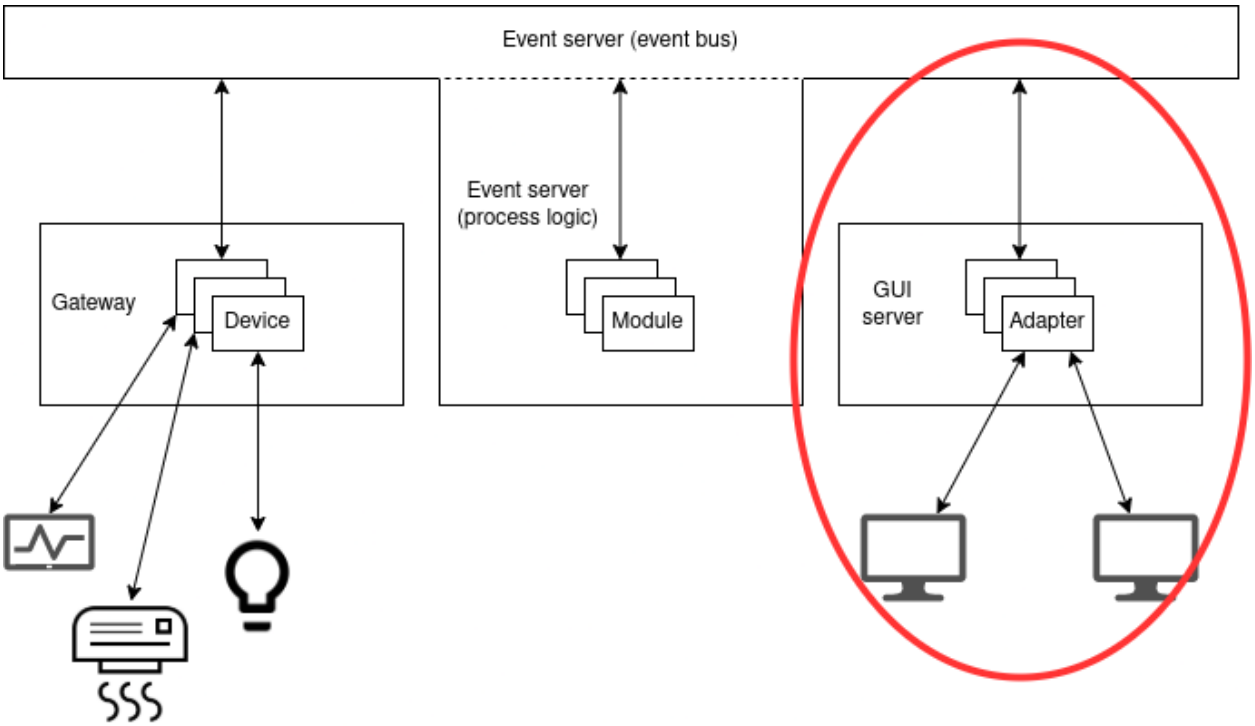
(continues on next page)

(nastavak sa prethodne stranice)

```
                        type=hat.event.server.common.EventPayloadType.JSON,
                        data=self._state)))]


class StateModuleSession(hat.event.server.common.ModuleSession):

    def __init__(self, module, group):
        self._module = module
        self._group = group

    @property
    def async_group(self):
        return self._group

    async def process(self, changes):
        # delegacija obrade dogadaja nazad modulu
        return self._module.module_process(changes)
```

Ovaj modul pretplacuje se na dogadaje koje registrira device i kreira svoje dogadaje koji sadrze stanje, u istom formatu kao i u prvom zadatku.

3.2.3 GUI server



Zadnja komponenta na koju se fokusiramo je GUI server, a ona sluzi za vizualizaciju podataka. Ona se, s jedne strane, spaja na event server i ukljucuje u interakciju s dogadajima, a, s druge, posluzuje HTTP servis na koji se korisnici mogu spojiti svojim web browserima i pregledati stanje sustava. Stanje sustava prezentira se kroz genericnu web aplikaciju, koja se moze konfigurirati da prikazuje preglede koje implementator sustava implementira. Aplikacija je implementirana u JavaScriptu i komunicira s GUI serverom pomocu WebSocket protokola, odnosno hat-juggler wrap-

pera. Konkretan format stanja koje GUI server salje klijentskoj aplikaciji propisuju specijalizirani moduli GUI servera, adapteri.

Komponenta se pokrece pozivom `hat-gui` u komandnoj liniji. Zadaje joj se argument `--conf` koji sadrzi putanju do JSON ili YAML konfiguracije koja sadrzi konkretne postavke. Format konfiguracije propisan je JSON shemom, a jedan minimalni primjer mogao bi biti:

```
---
type: gui
event_server_address: tcp+sbs://127.0.0.1:23012
address: http://0.0.0.0:23023
views:
  - name: login
    view_path: ./views/login
    conf_path: null
  - name: main
    view_path: ./views/main
    conf_path: null
initial_view: login
users:
  - name: user1
    password:
        hash: 0927f26c1e200037ef44e622d39d5b7c201690c85b9aa86545d6583ecff2b02f
        salt: 7af08c40f25d800fa3d1ab3f8199adbd
    roles:
        - user
    view: main
adapters:
  - module: adapters.state
    name: state
log:
    disable_existing_loggers: false
    formatters:
        default: {}
    handlers:
        console:
            class: logging.StreamHandler
            level: INFO
            stream: ext://sys.stdout
    root:
        handlers:
        - console
        level: INFO
    version: 1
...
```

Konfiguriraju se razne adrese, jedna za spajanje s event serverom, druga na kojoj GUI server posluzuje podatke... Konfiguriraju se i viewovi, to su spomenuti pregledi koje mi trebamo implementirati. Nakon toga slijedi konfiguracija korisnika. GUI server obavlja rudimentarno upravljanje korisnicima, u konfiguraciji se navode login podatci. Za lozinke se ocekuje da su hashirane SHA256 algoritmom, i da je „posoljen" s nasumicnim bajtima u `salt` polju i ponovno hashiran. Za potrebe nasih primjera, ove stvari ce biti hardkodirane, uvijek koristimo postavke iz ove konfiguracije, a za korisnika `user1` lozinka je `pass1`. Uz login podatke a korisnika se moze definirati koju ulogu (`role`) ima - na ovaj nacin moze se napraviti distinkcija izmedu administratora i obicnih korisnika te koji view korisnik vidi nakon sto se prijavi. Konacno, nakon korisnika ide konfiguracija specijaliziranih modula, adaptera. Vidimo slicnu strukturu kao i kod gatewayovih deviceva, zadaje se Python ime modula (npr. ovdje ce se u nekom trenutku zvati `import adapters.`

`state`) i ime adaptera koje ce se koristiti da identificira taj adapter u komunikaciji s event serverom i klijentskom aplikacijom.

Razvoj viewova, odnosno grafickih prikaza je tema za sebe koja ce biti pokrivena u odvojenom poglavlju. U ovom primjeru koristiti cemo gotovu, izbuildanu verziju viewova, a GUI server cemo samo konfigurirati da koristi te prikaze.

Glavni dio razvoja na serverskoj strani je implementacija adaptera. Pogledajmo sad sucelje koje ovi specijalizirani moduli moraju implementirati. Vidimo da on mora biti izveden kao odvojeni Python modul s globalnim varijablama `json_schema_id` i `json_schema_repo` te funkcijama `create_subscription` i `create_adapter`. Za globalne varijable vrijede iste primjedbe kao i kod deviceva i event server modula, to su opcionalne varijabe koje sluze za validaciju konfiguracije koja je postavljena te ce biti `None` u nasim primjerima. `create_subscription` treba moci primiti jedan argument, konfiguraciju adaptera, a vraca instancu hat.event.common.Subscription klase u kojoj se navodi na kakve tipove dogadaja se adapter pretplacuje. Konacno, `create_adapter` funkcija prima konfiguraciju adaptera i event klijent, a vraca instancu klase `hat.gui.common.Adapter`.

Ako pogledamo klasu `hat.gui.common.Adapter` vidimo da ona nasljeduje `hat.aio.Resource`, dakle, kao i devicevi i event server moduli, mora imati property `async_group`. Uz to sama adapterova klasa propisuje da mora postojati metoda `create_session` koja prima jedan argument tipa hat.gui.common.AdapterSessionClient. Slicno kao i moduli event servera, adapteri nece sami direktno komunicirati sa svojim klijetima, vec imaju sesije. U ovom kontekstu, jedna sesija predstavlja vezu na jednog klijenta koji je spojen na GUI server (mozemo to zamisliti kao da svaka otvorena sesija predstavlja jedan web browser koji je spojen na nas server). Vidimo da hat.gui.common.AdapoterSession zapravo nema nikakve dodatne metode i propertyje (osim, opet, `async_group` jer je instance `hat.aio.Resource`). To znaci da imamo slobodu bilo kako implementirati kako se tocno koristi `AdapterSessionClient` za komunikaciju s klijentima.

Preko `AdapterSessionClient`-a adapterova sesija komunicira s web aplikacijom u browseru. Vidimo da ona ima slicne metode i propertyje kao i juggler konekcija (jer nam GUI server zapravo predaje wrapper oko nje). Kod jugglera je ideja da povezuje dvije komunikacijske tocke s WebSocket protokolom. WebSocket se inace specijalizira za slanje poruka, a juggler nam pruza podrsku za neke dodatne funkcije. Jedna od tih funkcija je sinkronizacija stanja - vidimo da `Connection` (i `AdapterSessionClient`) ima propertyje `local_data` i `remote_data`. Jedna strana komunikacije moze u `local_data` zapisati bilo kakav JSON serijalizabilni objekt (preko metode `set_local_data`), i on ce se drugoj strani pojaviti u njenom `remote_data` propertyju. Uz to, moguce je raditi i obicno slanje poruka kroz `send` i `receive` metode, definirati RPC sucelja itd. Prakticno, u radu s jugglerom, odnosno `AdapterSessionClient`-om, najvise cemo se oslanjati na sinkronizaciju stanja i slanje poruka.

Sad imamo dovoljno informacija da napravimo jednostavnu implementaciju adaptera. Nastavljamo s nasim primjerom ranijeg zadatka s ampermetrima. Ako se sjecamo, napravili smo modul event servera koji registrira dogadaj tipa `('state')` ciji payload je dictionary gdje su kljucevi imena struja, a vrijednosti njihovi iznosi. Sad mozemo napraviti adapter koji ce to stanje propagirati do klijenata:

```
import hat.aio
import hat.event.common
import hat.gui.common
import hat.util


json_schema_id = None
json_schema_repo = None


async def create_subscription(conf):
    return hat.event.common.Subscription([('state', )])


async def create_adapter(conf, event_client):
    adapter = StateAdapter()
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    adapter._async_group = hat.aio.Group()
    adapter._event_client = event_client
    adapter._async_group.spawn(adapter._main_loop)
    adapter._sessions = set()

    return adapter


class StateAdapter(hat.gui.common.Adapter):

    @property
    def async_group(self):
        return self._async_group

    async def create_session(self, juggler_client):
        session = StateAdapterSession(
            juggler_client,
            self._async_group.create_subgroup())
        self._sessions.add(session)
        return session

    async def _main_loop(self):
        while True:
            events = await self._event_client.receive()
            for event in events:
                state = event.payload.data
                for session in self._sessions:
                    if session.is_open:
                        session.notify_state_change(state)


class StateAdapterSession(hat.gui.common.AdapterSession):

    def __init__(self, juggler_client, group):
        self._juggler_client = juggler_client
        self._async_group = group

    @property
    def async_group(self):
        return self._async_group

    def notify_state_change(self, state):
        self._juggler_client.set_local_data(state)
```

Ako ima nejasnoca vezanih uz poziv `spawn` metode, logika je ista kao i kod devicea, pa predlazemo da pogledate taj dio. Dakle, adapter u `_main_loop` ceka promjene stanja i kad primi dogadaj, proslijedi tu informaciju sesijama. Sesije onda dalje tu informaciju proslijede web klijentima.

Sad mozemo pokrenuti `hat-gui` s ovim adapterom, no jos uvijek nam fale viewovi. Kako smo spomenuli da je njihov razvoj odvojena tema za sebe, zasad cemo koristiti prethodno buildane resurse. Njima, i svim ostalim implementacijama koje smo radili u ovom poglavlju, mozete pristupiti na ovom linku Pokretanjem svega (`hat-event`, `hat-gateway` i `hat-gui`) i otvaranjem adrese 127.0.0.1:23023 trebala bi se otvoriti prvo login stranica (user1, pass1), a nakon logina

prikaz u kojem se vidi JSON reprezentacija stanja koje smo propagirali kroz dogadaje. Iduce vece poglavlje bavit ce se osnovama razvoja grafickog sucelja u bibliotekama iz hat-open projekta.

POGLAVLJE 4

Modbus hat aplikacija

U proslom zadatku vidjeli smo kako se spojiti Modbus protokolom na termometar kroz jednostavnu Pyhton skriptu. U ovom dijelu implementiramo slicnu stvar, koristenjem komponenti iz hat-open projekta. Poglavlje o Hatu temeljito opisuje pojedine komponente a ovdje cemo ih primjeniti za rjesavanje problema s termometrom.

4.1 Repozitorij

Krecemo s repozitorijem hat-quickstart. Ovo je template repozitorij na temelju kojeg se mogu kreirati novi repozitoriji (*Use this template* gumb). U repozitoriju su napisane upute kako postaviti razvojno okruzenje. Ono bi trebalo raditi na Linuxu, kod ostalih operacijskih sustava mozda naidete na probleme i u tom slucaju preporucujemo rad ili kroz virtualnu masinu ili kroz docker.

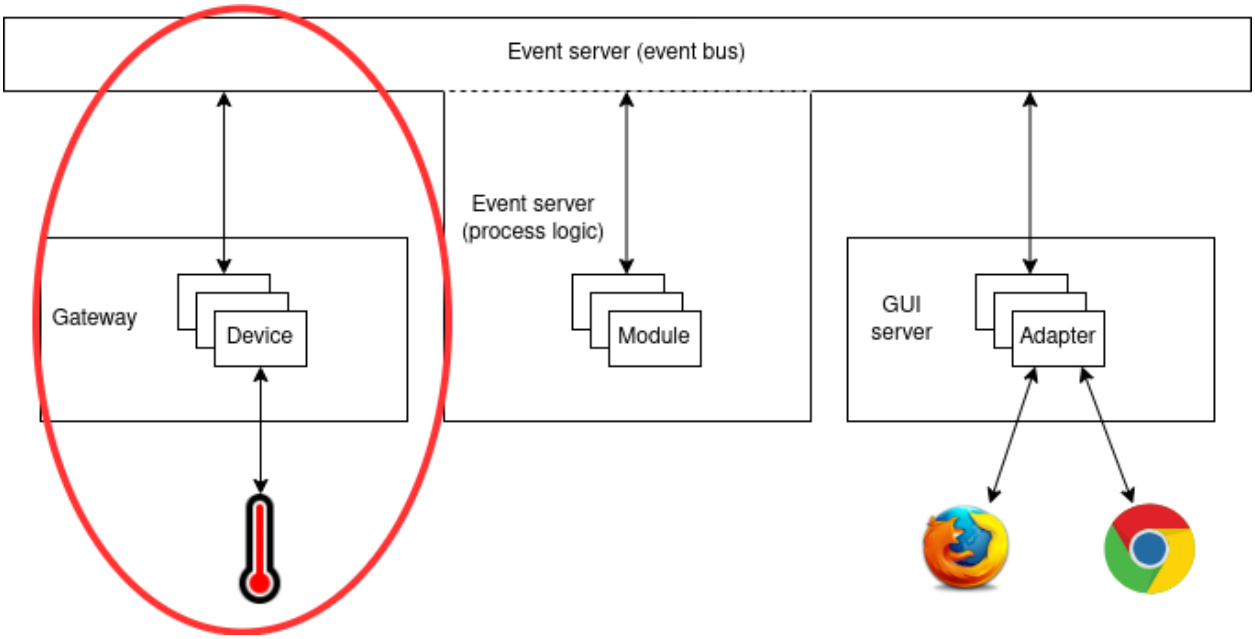
Repozitorij ima par bitnih direktorija. Prvi su `src_py` i `src_js`, u kojima ce se nalaziti implementacija nase aplikacije. Python dio implementacije fokusirat ce se na ocitanje i obradu podataka, a JavaScript na vizualizaciju kroz web sucelje. U `playground` direktoriju imamo razlicite pomocne skripte i konfiguracije pomocu kojih se sustav moze pokretati. Ostale datateke i direktoriji se takoder koriste, ali na ove cemo se uglavnom fokusirati tijekom rada na ovom zadatku. Pozicioniranjem u direktorij `playground/run` mozemo pokrenti skriptu `system.sh` (ili `.bat` za Windowse), otvoriti `localhost:23023` i vidjeti minimalno graficko sucelje s jednim brojacem.

Bacimo li dublji pogled na to sto se tocno dogada pozivom `system` skripte, vidimo da ona pokrece `hat-orchestrator` komponentu. Ova komponenta sluzi tome da paralelno starta proizvoljni broj drugih procesa. Pogledamo li njenu konfiguraciju, u `playground/run/data/orchestrator.yaml`, vidimo u `components` polju koji procesi se sve pokrecu. Vidimo da su to druge komponente iz hat-open projekta (pogledajte predavanje 3.1. za kratki pregled komponenti koje postoje i neke njihove generalne svrhe). Za rjesenje problema, implementirati cemo jedan device, event server modul i adapter. Takoder cemo prilagoditi view da ima tocniji ispis, tj. da ne pise *counter* vec temperature.

4.2 Rjesenje

Ovdje opisujemo pristup koji uzimamo kad rjesavamo problem, komponentu po komponentu. Krecemo od devicea koji ce nam komunicirati s modbus uredajem, nakon toga implementiramo event server modul, koji ce obavljati izracun temperature (djeljenje s 10 jer protokol salje vrijednost pomnozenu s 10), iza toga adapter koji ce pripremati te podatke za vizualizaciju, i konacno view kojeg cemo prilagoditi da prikazuje podatke. U quickstart repozitoriju vec postoje primjeri za svaki od ovih tipova specijaliziranih modula, tako da cemo se donekle oslanjati samo na njihovu modifikaciju.

4.2.1 Device



Zelimo implementirati device koji ce svakih nekoliko sekundi slati Modbus zahtjev za citanje na termometar i registrirati dogadaj s informacijom koja je primljena preko protokola. Slicnu funkcionalnost smo vec implementirali u drugom zadatku, samo sad ju trebamo upakirati u implementaciju devicea i umjesto ispisa na konzolu, registrirati dogadaj. Implementacija uredaja u `src_py/project/devices/example.py` je u principu dovoljno dobra, glavna modifikacija koju trebamo napraviti je prilagodba countera. *Example* verzija uredaja se ne spaja na nista nego samo ima jedan interni brojac kojeg inkrementira svakih nekoliko sekundi. Umjesto toga, mi se zelimo spojiti na termometar i svakih nekoliko sekundi registrirati ocitanje s njega. Takoder, putanja `project/devices/example` je malo nejasna, tako da cemo ju preimenovati u nesto sto ima smisla za nas projekt: `workshop/devices/modbus`.

Primjetimo da sad vise ne mozemo pozvati sustav kroz `system.sh/bat`. Razlog tome je upravo ova promjena putanje, device koji konfiguriramo u gatewayu ima staro Python ime, pa cemo ga prilagoditi u `module: workshop.devices.modbus` (u `playground/run/data/gateway.yaml`). Uz to, dodatno cemo prilagoditi konfiguraciju tako da damo konkretnija imena deviceu i gatewayu: `modbus1` i `gateway1`.

Device sad izgleda ovako:

```
import asyncio
import hat.aio
import hat.event.common
import hat.gateway.common
from hat.drivers import modbus, tcp
```

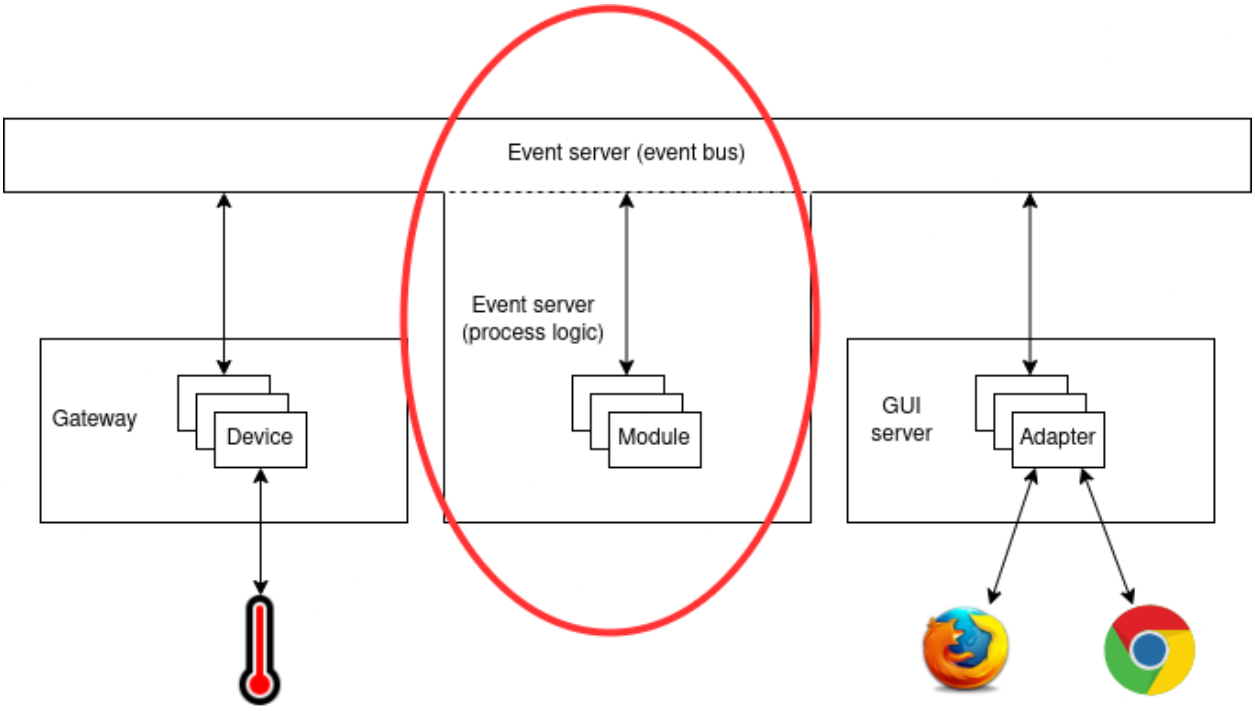
(continues on next page)

(nastavak sa prethodne stranice)

```
json_schema_id = None
json_schema_repo = None
device_type = 'modbus'


async def create(conf, event_client, event_type_prefix):
    device = ModbusDevice()

    device._async_group = hat.aio.Group()
    device._event_client = event_client
    device._event_type_prefix = event_type_prefix
    device._task = asyncio.create_task(device._main_loop())

    return device


class ModbusDevice(hat.gateway.common.Device):

    @property
    def async_group(self):
        return self._async_group

    async def _main_loop(self):
        modbus_type = modbus.ModbusType.TCP
        address = tcp.Address('161.53.17.239', 8502)
        master = await modbus.create_tcp_master(modbus_type, address)
        while True:
            data = await master.read(
                device_id=1,
                data_type=modbus.DataType.HOLDING_REGISTER,
                start_address=4003, quantity=1)
            self._event_client.register([
                hat.event.common.RegisterEvent(
                    event_type=(*self._event_type_prefix,
                                'gateway', '4003'),
                    source_timestamp=None,
                    payload=hat.event.common.EventPayload(
                        type=hat.event.common.EventPayloadType.JSON,
                        data=data[0]))])
```

Trebalo bi se moci pokrenuti, no sad smo uveli promjene zbog kojih osnovni quickstart primjer nece moci samostalno raditi. Ali, to ce se promijeniti kad prilagodimo ostale komponente.

4.2.2 Modul



Sad se prebacujemo na event server module. Ako pogledamo koji moduli postoje, vidjet cemo direktorij `src_py/project/modules` s modulima `example.py` i `enable_all.py`. `enable_all` mozemo ignorirati, njegova svrha je da registrira dogadaje za paljenje deviceova (vise info u poglavlju o Hatu).

Odmah cemo oba modula prebaciti u `workshop/modules`, a `example` cemo preimenovati u `temperature` jer cemo ga preinaciti u to da racuna temperaturu na temelju dogadaja koje registrira device.

Znamo da device registrira ocitanja temperatura u dogadajima s tipom `('gateway', 'gateway1', 'modbus', 'modbus1', 'gateway', '4003')`, tako da cemo pretplatiti modul na taj tip. Nakon toga, nema neke potrebe za vecim preinakama, osim prilagodbe `process` metode u sesiji od modula. Ona ce sad registrirati dogadaj s tipom `('temperature')` a payload ce joj biti tocna temperatura, iznos koji primi preko Modbusa podijeljen s 10.

Implementacija modula izgleda ovako:

```
import hat.aio
import hat.event.server.common


json_schema_id = None
json_schema_repo = None

_source_id = 0


async def create(conf, engine):
    module = TemperatureModule()

    global _source_id
    module._source = hat.event.server.common.Source(
        type=hat.event.server.common.SourceType.MODULE,
```

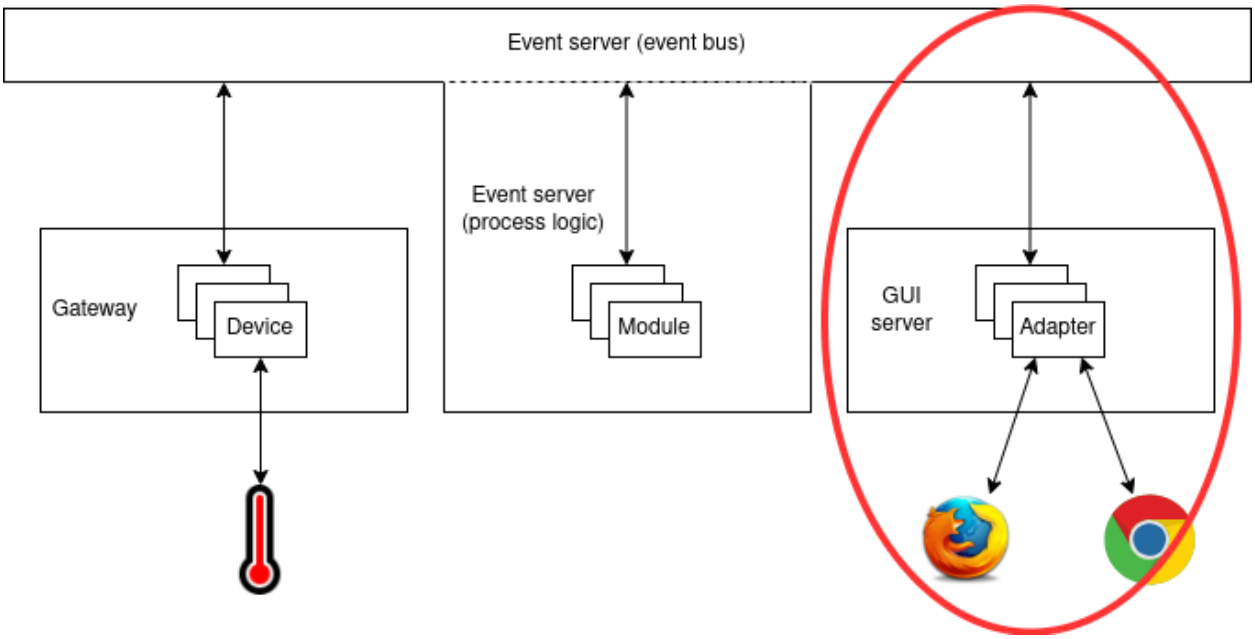
(continues on next page)

(nastavak sa prethodne stranice)

```
        name=__name__,
        id=_source_id)
    _source_id += 1

    module._subscription = hat.event.server.common.Subscription([
        ('gateway', '?', 'modbus', '?', 'gateway', '4003')])
    module._async_group = hat.aio.Group()
    module._engine = engine

    return module


class TemperatureModule(hat.event.server.common.Module):

    @property
    def async_group(self):
        return self._async_group

    @property
    def subscription(self):
        return self._subscription

    async def create_session(self):
        return TemperatureModuleSession(self._engine, self._source,
                                        self._async_group.create_subgroup())


class TemperatureModuleSession(hat.event.server.common.ModuleSession):

    def __init__(self, engine, source, group):
        self._engine = engine
        self._source = source
        self._async_group = group

    @property
    def async_group(self):
        return self._async_group

    async def process(self, changes):
        return [
            self._engine.create_process_event(
                self._source,
                hat.event.server.common.RegisterEvent(
                    event_type=('temperature', ),
                    source_timestamp=None,
                    payload=hat.event.common.EventPayload(
                        type=hat.event.common.EventPayloadType.JSON,
                        data=f'{event.payload.data / 10}')))
            for event in changes]
```

Ne zaboravimo da je i dalje potrebno uvesti promjene u konfiguraciju, analogne promjenama u gatewayu, dakle imena modula.

4.2.3 Adapter



Konacno se fokusiramo na vizualizaciju. Prvi korak je implementacija adaptera koji bi preko svog sucelja za komunikaciju s web klijentima posluzivao stanje u kojem je ocitanje temperature. On bi se pretplacivao na `('temperature')` dogadaj koji registriramo u event serverovom modulu.

Ako pogledamo adaptere iz quickstarta, vidimo da postoji `src_py/project/adapters/example.py`. Njega cemo prilagoditi da ne radi vise s brojacem, vec da se pretplacuje na promjene temperature i prosljeduje ih svojim sesijama. To postizemo implementacijom `create_subscription` funkcije. Drugi korak je proslijedivanje temperature sesijama, sto se dogadalo u `_main_loop` metodi. Ona zapravo moze biti ista, ali malo cemo prilagoditi imena varijabli, nema potrebe da se temperatura pohranjuje u privatnim varijablama adapterove klase itd. Tako dolazimo do sljedece implementacije:

```
import hat.aio
import hat.event.common
import hat.gui.common
import hat.util


json_schema_id = None
json_schema_repo = None


async def create_subscription(conf):
    return hat.event.common.Subscription([('temperature', )])


async def create_adapter(conf, event_client):
    adapter = TemperatureAdapter()

    adapter._async_group = hat.aio.Group()
    adapter._event_client = event_client
    adapter._state_change_cb_registry = hat.util.CallbackRegistry()
```

(continues on next page)

(nastavak sa prethodne stranice)

```
    adapter._sessions = set()

    adapter._async_group.spawn(adapter._main_loop)

    return adapter


class TemperatureAdapter(hat.gui.common.Adapter):

    @property
    def async_group(self):
        return self._async_group

    async def create_session(self, juggler_client):
        session = TemperatureAdapterSession(
            juggler_client,
            self._async_group.create_subgroup())
        self._sessions.add(session)
        return session

    async def _main_loop(self):
        while True:
            events = await self._event_client.receive()
            for event in events:
                temperature = event.payload.data
            for session in self._sessions:
                if session.is_open:
                    session.notify_state_change(temperature)


class TemperatureAdapterSession(hat.gui.common.AdapterSession):

    def __init__(self, juggler_client, group):
        self._juggler_client = juggler_client
        self._async_group = group

    @property
    def async_group(self):
        return self._async_group

    def notify_state_change(self, state):
        self._juggler_client.set_local_data(state)
```

Opet cemo promijeniti putanju adapteru tako da ga prebacimo u `workshop/adapter` i preimenujemo u `temperature.py` (nema veze sto je isto kao i modul, sama cinjenica da su u drugim direktorijima je dovoljna distinkcija), pa je potrebno prilagoditi i konfiguraciju. Osim promjene putanje, prilagoditi cemo i ime adaptera iz `adapter` u `temperature`.

4.2.4 View

Da bi prilagodba bila kompletna, potrebno je prilagoditi i view. Dosad nismo toliko zalazili u detalje kako se view implementira, ali nam za pocetak oni nisu ni previse bitni. Implementacija viewa je u `src_js/views/main/index.js`. Vidimo da vec postoji neka implementacija koja ima funkciju `vt` koja vraca listu. Lista sadrzi ime taga i njegov sadrzaj. Za graficke prikaze koristimo bibilioteke koje ovakve strukture podataka pretvaraju u DOM (Document Object Model). Tako kad implementiramo view, ono sto vrati njegova funkcija proslijeduje se tim bibliotekama i one generiraju DOM s elementima:

```
<span>counter: 10</span>
```

Ocita je jos jedna promjena - izraz `r.get('remote', 'adapter', 'counter')` se pretvoro u broj 10. U viewovima preko varijable `r` pristupamo renderer objektu (dokumentacija je nekompletna, bolje mozda gledati kod). Cijela ideja iza renderera je da on ima neko svoje stanje i na temelju tog stanja generira DOM. Kad se stanje promijeni, ponovno se pokrene izracun DOM-a na temelju tog novog stanja. Stanju pristupamo preko funkcije `r.get`, a mozemo ga mijenjati preko funkcije `r.set`. Tako kad kazemo `counter: r.get('remote', 'adapter', 'counter')`, zapravo oznacavamo da iza dvotocke pise vrijednost procitana iz stanja aplikacije. Argumentima poslanim `r.get` funkciji odredujemo putanju do dijela stanja koji nas zanima. Konkretno u ovom slucaju, brojacu pristupamo s putanjom `remote/adapter/counter` jer je stanje objekt:

```
{ remote: { adapter: { counter: 10 } } , local: {}, xyz: 100, ... }
```

Kod viewova postoji dodatni aspekt rada sa stanjem, a to je cinjenica da je nasa aplikacija spojena na GUI server. GUI server notificira promjene stanja svojim klijentima. To znaci da se u nekom trenutku implicitno poziva `r.set` kad neki adapter prijavi da mu se stanje promijenilo. Konkretno, dio stanja na koji utjece adapter nalazi se na putanji `remote/<ime_adaptera>`.

Vracajuci se na radionicu, dosadasnji view s brojacem nam vise ne odgovara, htjeli bismo da umjesto counter pise temperature. Dodatno, mijenjali smo ime i strukturu stanja adaptera, pa vise ni `r.get` nije precizan. Ako zelimo promijeniti tekst, prevodimo `counter:` u `temperature:`, a nova putanja je `remote/temperature` (`temperature` je ime adaptera, a njegovo stanje je samo jedan broj).

Ta implementacija sad bi trebala izgledati ovako:

```
import 'main/index.scss';


export function vt() {
    return ['span', `temperature: ${r.get('remote', 'temperature')}`];
}
```

Nakon buildanja viewa (`doit js_view`), mozemo otvoriti browser na adresu `http://127.0.0.1:23023` i vidjeti nas novi termometar.