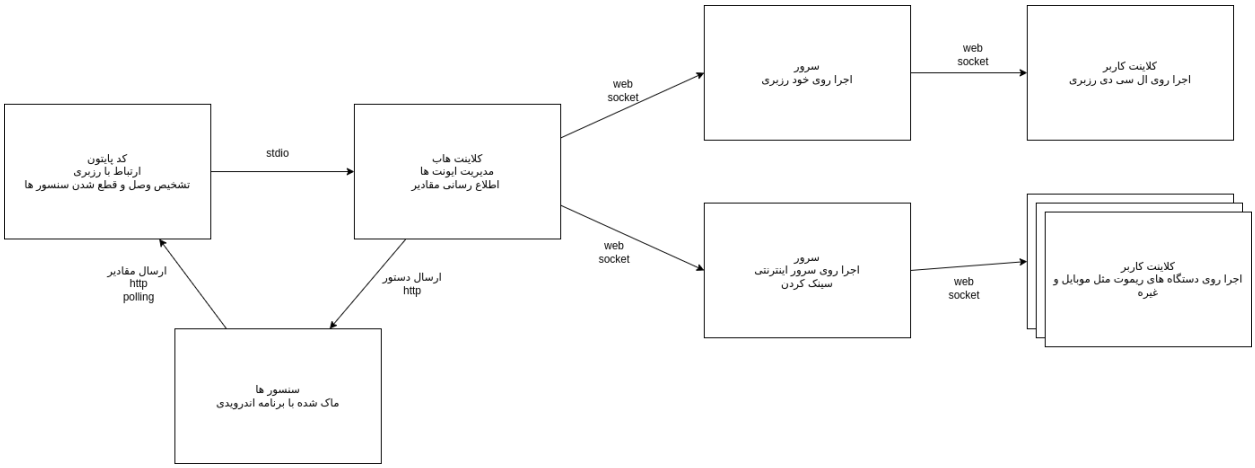
بسمه تعالی

آزمایشگاه سخت افزار - پروژه هاب مرکزی خانه هوشمند
تیم 3 - امیر نژادملایری، حمیدرضا کلباسی، صالح صفرنژاد
گزارش نهایی

در این پروژه ما تلاش کردیم تا یک هاب خانه هوشمند روی یک دستگاه Raspberrypi 3 ایجاد کنیم که بتواند حسگر ها (سنسور ها) و عملگر ها (اکتویتور ها) را در یک خانه هوشمند به وسیله شبکه وای فای کنترل کند و به کاربر این امکان را بدهد تا هم از طریق صفحه نمایشی که روی رزبری قرار گرفته و هم از طریق دستگاه هایی که از طریق اینترنت متصل می شوند بتواند وضعیت خانه هوشمندش را ببیند یا آن را تغییر دهد و یا این که برای آن برنامه ریزی کند.

طرح کلی:

در شکل زیر می توانید طرح کلی قسمت ها و ارتباط آن ها با هم را مشاهده کنید. در ادامه جزییات هر قسمت توضیح داده خواهد شد.



کد پایتون تشخیص وصل شدن و قطع شدن سنسور ها:
در این قسمت ما به وسیله بررسی جدول آرپ متوجه تغییرات در لیست آی پی های دستگاه های متصل می شویم و لیست دستگاه های جدیدا متصل شده یا جدیدا قطع شده را به کلاینت هاب اطلاع رسانی می کنیم

```python
def update_devices(arp_df):
    available_ips = set(arp_df.loc[(arp_df['HWaddress'] != '(incompatible)') & (arp_df['Iface'] == 'wlan0'),'Address'])
    for ip in available_ips:
        if ip not in ip_table:
            on_new_device_connected(ip)
    disconnected_devices = []
    for ip in ip_table:
        if ip not in available_ips:
            disconnected_devices.append(ip)
            on_device_disconnected(ip)
    for ip in disconnected_devices:
        ip_table.remove(ip)
```

در یک ترد دیگر، دیوایس هایی که مطمئن هستیم متصل هستند را پول می کنیم تا مقادیر سنسور هایشان را به دست آوریم:

```python
class DevicePollerThread(threading.Thread):
    def run(self):
        while True:
            sleep(0.7)
            with lock:
                for d in devices:
                    r = requests.get(url=f"http://{d['ip']}:8080{d['uri']}")
                    if d["value"] != r.text:
                        d["value"] = r.text
                        send({ "type": "update_device", "value": d })
```

تابع سند یک جیسون را در stdout پرینت می کند تا کلاینت هاب از آن استفاده کند.

کلاینت هاب:

بعد از تشخیص سنسور ها توسط قسمت بالا، اطلاعات به این قسمت ارسال می شود. در این قسمت، یک event loop وجود دارد که در آن، ابتدا بررسی می شود که آیا پیغام جدیدی از کلاینت های دیگر آمده است یا نه (پیغام ها شامل افزودن قواعد جدید، اعمال عمل برای عملگر ها، ... است) و اگر آمده بود ابتدا آن ها بررسی می شود. سپس ورودی های آمده از کد پایتونی (شامل اضافه شدن دستگاه جدید، حذف دستگاه یا تغییر مقدار یک سنسور) بررسی می شوند و در صورت نیاز (یعنی در صورت این که مقدار گزارش شده با استیت فعلی ما فرق می کرد. توجه کنید که سنسور ها به صورت poll بررسی می شوند و مقدار یک سنسور حتی در صورت تغییر نکردن نیز به دست ما می رسد ولی ما برای صرفه جویی در پهنای باند سرور و کاربران ریموت فقط در صورت تغییر ایونت آپدیت ارسال می کنیم) ایونت های آپدیت ارسال می شود.

```rust
        _python_stdout.read_line(&mut x).unwrap();
        let device_event: DeviceEvent = serde_json::from_str(&x).unwrap();
        match device_event {
            DeviceEvent::HeartBeat => (),
            DeviceEvent::NewDevice { mut value } => {
                if value.kind == DeviceKind::Actuator {
                    value.value = match value.name.as_str() {
                        "curtain" => "close",
                        "outlet" => "off",
                        _ => "",
                    }
                    .to_owned();
```

```
                }
                state
                    .devices
                    .entry(value.location.clone())
                    .or_default()
                    .push(value);
            }
            DeviceEvent::UpdateDevice { value } => {
                for d in
state.devices.get_mut(&value.location).unwrap().iter_mut() {
                    if d.uri == value.uri {
                        d.value = value.value.clone();
                    }
                }
            }
        }
```

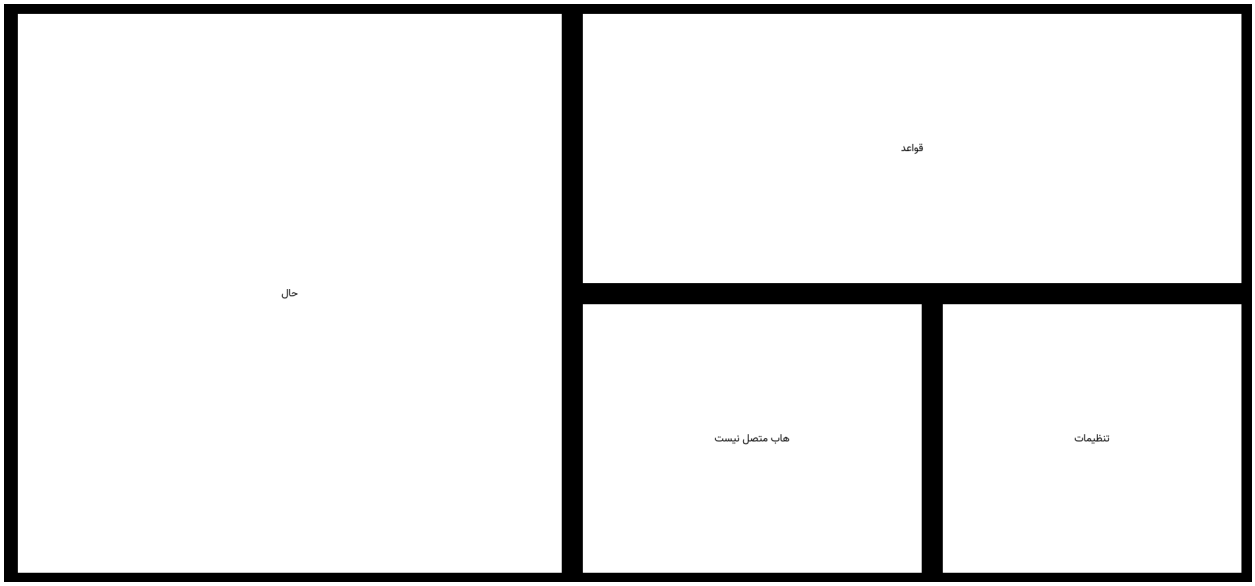
سپس قواعد بررسی می شوند و در صورتی که هر کدام از آنها به مرحله اجرا رسیده باشند، کد مربوط به آن ها اجرا می شود:

```rust
impl State {
    fn check_rules(&mut self) {
        let mut rules: Vec<(RuleState, Job)> = mem::take(dest: &mut self.rules);
        rules.retain(|x: &(RuleState, Job)| match x.0 {
            RuleState::Timer { count: usize, .. } => count != 0,
            RuleState::Sensor { .. } => true,
        });
        for (rule: &mut RuleState, job: &mut Job) in &mut rules {
            match rule {
                RuleState::Timer {
                    next: &mut SystemTime,
                    interval: &mut Duration,
                    count: &mut usize,
                } => {
                    if *next < SystemTime::now() {
                        *next = *next + *interval;      You, 2 days ago • Add basics of handler.py …
                        job.do_it();
                        *count -= 1;
                    }
                }
                RuleState::Sensor {
                    operator: &mut Operator,
                    location: &mut String,
                    device: &mut String,
                    value: &mut i32,
                    is_sat: &mut bool,
                } => {
                    let condition_result: bool = self&mut State
                        .check_condition(&location, &device, *value, *operator)Option<bool>
                        .unwrap_or(default: false);
                    if *is_sat != condition_result {
                        *is_sat = condition_result;
                        if *is_sat {
                            job.do_it();
                        }
                    }
                }
            }
        }
        self.rules = rules;
    }fn check_rules
```

سرور:

سرور به این صورت عمل می کند که تعدادی اتاق دارد و در هر اتاق تعدادی کلاینت عادی و یک کلاینت هاب را می پذیرد. سرور با تمام کلاینت ها ارتباط وب سوکتی برقرار می کند (به این وسیله از سربار پهنای باند ساخت کانکشن و پولینگ جلوگیری می شود) و کلاینت ها می توانند از طریق سرور به کلاینت هاب دستوراتی ارسال کنند و کلاینت هاب می تواند از طریق سرور به کلاینت های دیگر آپدیت اطلاعات سنسور ها را ارسال کند.

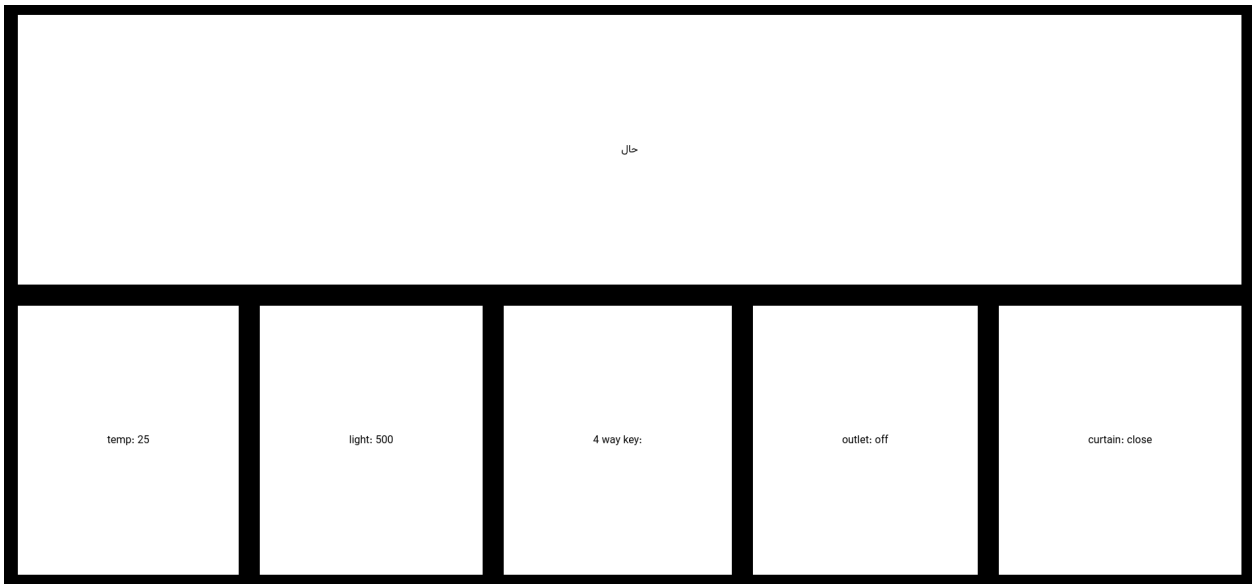
هر وقت که کلاینت هاب به هر دلیلی قطع شود (که این اتفاق در حالت سرور ریموت ممکن است بیفتد) سرور یک پیغام آپدیت به بقیه کلاینت ها ارسال می کند و آن ها به کاربرشان نمایش می دهند که کلاینت هاب متصل نیست. در این حالت اگر کاربر تلاش کند تا یک فرمان صادر کند، کلاینتش به او یک پیغام خطا نمایش می دهد.



به این وسیله همواره تضمین می شود که وضعیت مشاهده شده توسط کاربر با وضعیت واقعیت همواره سینک باشد و کاربر روی اطلاعات نادرستش فرمانی صادر نکند.

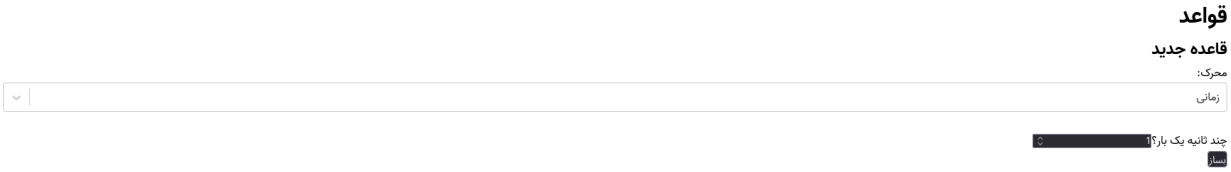
کلاینت کاربر:

کلاینت کاربر رابط کاربری سیستم ما با کاربر است. این رابط کاربری که با جاوا اسکریپت و React نوشته شده است، با وب سوکت همان گونه که در قسمت سرور مطرح شد به سرور متصل می شود و از طریق سرور با کلاینت هاب ارتباط برقرار می کند. در صفحه اصلی کاربر لیست مکان های خودش را می بیند و بعد از انتخاب هر مکان با صفحه ای مشابه عکس زیر مواجه می شود:



کاربر در این صفحه می تواند مقادیر سنسور های موجود در این اتاق را ببیند و با کلیک بر روی اکتویتور ها مقادیر آن ها را عوض کند (مثلا پرده را از باز به بسته تغییر دهد یا پریز را بین خاموش و روشن جا به جا کند).

هم چنین کاربر می تواند بخش قواعد را انتخاب کند و قاعده هایی که می خواهد را ثبت کند:



کاربر می تواند از بین محرک های زمان بندی و تغییر وضعیت سنسور ها یکی را انتخاب کند که تغییر وضعیت سنسور ها عملگر کوچکتر، بزرگتر و مساوی را پشتیبانی می کند. هر قاعده یک عملکرد نیز دارد که می تواند انجام یک عمل روی اکتویتور باشد یا در آوردن صدای یک بوق کوچک روی رزبری باشد که متاسفانه چون در قسمت رابط کاربری عملکرد پیاده سازی نشده کاربر فقط می تواند از بوق استفاده کند. نوع عملکرد ها را در کد کلاینت هاب می توانید ببینید:

```rust
#[derive(Debug, Serialize, Deserialize)]
#[serde(tag = "type")]
#[serde(rename_all = "snake_case")]
4 implementations
enum Job {
    Beep,
    Actuator { location: String, device: String, act: String }
}

impl Job {
    fn do_it(&self, state: &mut State) {
        match self {
            Job::Beep => println!("\x07"),
            Job::Actuator { location: &String, device: &String, act: &String } => {
                let device: &mut Device = state &mut State
                        .devices HashMap<String, Vec<Device>>
                        .get_mut(&location)? &mut Vec<Device>
                        .iter_mut() IterMut<'_, Device>
                        .find(|x: &&mut Device| x.name == device)?;
                let uri: String = format!("http://{}:8080{}/{act}", device.ip, device.uri);
                device.value = value;
                reqwest::blocking::get(url: uri).unwrap();
            },
        }
    }
}
```

بخش نصب شده روی رزبری:

این بخش به سرور لوکال رزبری وصل می شود و صفحه آن روی ال سی دی لمسی رزبری (که متاسفانه ما نتوانستیم آن را دریافت کنیم) نمایش داده خواهد شد. چون این بخش به سرور لوکال رزبری وصل می شود، پیغام «هاب متصل نیست» را هرگز نمی بیند.

بخش ریموت:

این بخش به سروری که روی اینترنت است متصل می شود ولی دقیقا همان کد قسمت بالا را اجرا می کند. کاربر می تواند به سایت اینترنتی ما برود و از طریق مرورگر وب خودش با لپ تاپ، موبایل، تلوزیون یا هر دستگاه دیگری که از مرورگر وب پشتیبانی می کند هاب خانه هوشمند خودش را کنترل کند. هم چنین در حالت موبایل کاربر می تواند آن را به صورت یک اپ موبایل PWA روی گوشی های اندروید و IOS نصب کند.

شبیه‌ساز حسگرها و عملگرها:

برای شبیه‌سازی رفتار حسگرها (sensor) و عملگرها (actuator) در ارتباط با هاب مرکزی خانه هوشمند یک برنامه اندروید به زبان جاوا ساخته شده که از طریق لینک زیر می‌توان کد این برنامه را مشاهده کرد (کد ها به پیوست نیز ارسال شده است):

https://github.com/HKalbasi/hardware_lab_smart_home_hub/tree/main/Emulator

صفحه نمایش:

در قسمت بالای این صفحه می‌توان نام اتاق مورد نظر را تنظیم کرد. در زیر این قسمت در سمت راست حسگر ها و در سمت چپ عملگر ها قرار دارند که با استفاده از کلید دو حالته موجود می‌توان هر کدام از آن ها را فعال یا غیر فعال کرد. همانطور که قابل مشاهده است هر دستگاه حاوی برنامه شبیه‌ساز متعلق به یک اتاق و نمایش دهنده هر ترکیب دلخواه از حسگرها و عملگرها باشد.

اطلاعات اتاق:

برای ذخیره سازی و ارسال اطلاعات وسایل شبیه‌سازی شده از دو کلاس Room و Sensor استفاده شده است. (به جهت مشابه بودن نوع اطلاعات عملگرها و حسگرها و جلوگیری از کد مشابه، برای عملگرها هم از کلاس Sensor استفاده شده است)

```java
public class Sensor {
    private final String type;
    private final String uri;

    public Sensor(String type, String uri) {
        this.type = type;
        this.uri = uri;
    }

    public String getType() { return type; }

    public String getUri() { return uri; }
}
```

```java
public class Room {
    private final String room;
    private final ArrayList<Sensor> sensors;
    private final ArrayList<Sensor> actuators;

    public Room(String room, ArrayList<Sensor> sensors, ArrayList<Sensor> actuators) {
        this.room = room;
        this.sensors = sensors;
        this.actuators = actuators;
    }

    public String getRoom() { return room; }

    public ArrayList<Sensor> getSensors() { return sensors; }

    public ArrayList<Sensor> getActuators() { return actuators; }
}
```

هر شیء از کلاس Room حاوی نام اتاق، لیست حسگرها و لیست عملگرها است. همچنین هر شیء از کلاس Sensor دارای نوع که بیانگر یکی از شش حسگر و عملگر تعریف شده است و uri که به آن پاسخ می‌دهد می‌باشد.

سرور شبیه‌ساز:

نحوه کار سه عملگر و حسگرهای دما و روشنایی بدین صورت است که با دستور هاب کار می‌کنند، یعنی حسگرهای گفته شده هر زمان که هاب درخواست دهد مقدار فعلی را در پاسخ برای هاب می‌فرستند، همچنین عملگرها نیز با درخواست هاب تغییر وضعیت می‌دهند.

```java
public class EmulatorServer extends NanoHTTPD {
    private FirstFragment fragment;

    public EmulatorServer() throws IOException {
        super( port: 8080);
        start(NanoHTTPD.SOCKET_READ_TIMEOUT, daemon: false);
    }

    public void setFragment(FirstFragment fragment) { this.fragment = fragment; }

    public Response serve(IHTTPSession session) {
        String response = "";
        switch (session.getUri()) {
            case "/conf":
                response += fragment.getConf(session.getRemoteIpAddress());
                break;
            case "/4way_key/key1/on":
                fragment.keyOn( keyNum: 1);
                break;
            case "/4way_key/key2/on":
                fragment.keyOn( keyNum: 2);
                break;
            case "/4way_key/key3/on":
                fragment.keyOn( keyNum: 3);
                break;
            case "/4way_key/key4/on":
                fragment.keyOn( keyNum: 4);
                break;
```

پیاده سازی این سرور با استفاده از کتابخانه NanoHTTPD انجام شده، بدین صورت که همزمان با ساخته شدن شیء EmulatorServer یک سرور روی پورت 8080 دستگاه راه‌اندازی می‌شود که با دریافت درخواست تابع serve را صدا می‌زند.

درخواست اطلاعات دستگاه:

```
case "/conf":
    response += fragment.getConf(session.getRemoteIpAddress());
    break;
```

```
public String getConf(String ip) {
    hubIp = "http://" + ip + ":8080";

    ArrayList<Sensor> s = new ArrayList<>();

    if (binding.tempSwitch.isChecked()) {
        s.add(new Sensor( type: "temp",  uri: "/temp"));
    }
    if (binding.lightSwitch.isChecked()) {
        s.add(new Sensor( type: "light",  uri: "/light"));
    }
    if (binding.moveSwitch.isChecked()) {
        s.add(new Sensor( type: "move",  uri: "/move"));
    }

    ArrayList<Sensor> a = new ArrayList<>();

    if (binding.keySwitch.isChecked()) {
        a.add(new Sensor( type: "4 way key",  uri: "/4way_key"));
    }
    if (binding.outletSwitch.isChecked()) {
        a.add(new Sensor( type: "outlet",  uri: "/outlet"));
    }
    if (binding.curtainSwitch.isChecked()) {
        a.add(new Sensor( type: "curtain",  uri: "/curtain"));
    }

    return gson.toJson(new Room(room, s, a));
}
```

سرور با دریافت درخواست /conf ابتدا آدرس ip هاب را ذخیره می‌کند و سپس با توجه حسگرها و عملگرهای فعال پاسخ را مطابق شکل بالا در پاسخ برمی‌گرداند.

کلید چهاپل:

```
public void keyOn(int keyNum) {
    requireActivity().runOnUiThread(() -> {
        switch (keyNum) {
            case 1:
                binding.key1.setVisibility(View.INVISIBLE);
                binding.key12.setVisibility(View.VISIBLE);
                break;
            case 2:
                binding.key2.setVisibility(View.INVISIBLE);
                binding.key22.setVisibility(View.VISIBLE);
                break;
            case 3:
                binding.key3.setVisibility(View.INVISIBLE);
                binding.key32.setVisibility(View.VISIBLE);
                break;
            case 4:
                binding.key4.setVisibility(View.INVISIBLE);
                binding.key42.setVisibility(View.VISIBLE);
                break;
        }
    });
}
```

```
case "/4way_key/key1/on":
    fragment.keyOn( keyNum: 1);
    break;
case "/4way_key/key2/on":
    fragment.keyOn( keyNum: 2);
    break;
case "/4way_key/key3/on":
    fragment.keyOn( keyNum: 3);
    break;
case "/4way_key/key4/on":
    fragment.keyOn( keyNum: 4);
    break;
case "/4way_key/key1/off":
    fragment.keyOff( keyNum: 1);
    break;
case "/4way_key/key2/off":
    fragment.keyOff( keyNum: 2);
    break;
case "/4way_key/key3/off":
    fragment.keyOff( keyNum: 3);
    break;
case "/4way_key/key4/off":
    fragment.keyOff( keyNum: 4);
    break;
```

برای روشن یا خاموش کردن هریک از کلیدهای کلید چهارپل مطابق شکل سمت راست شماره کلید و اینکه باید خاموش یا روشن شود را از uri استخراج کرده و مطابق شکل سمت چپ تغییرات را اعمال می‌کند.

پریز هوشمند:

```
public void outletOn() {
    requireActivity().runOnUiThread(() -> {
        binding.outlet.setVisibility(View.INVISIBLE);
        binding.outlet2.setVisibility(View.VISIBLE);
    });
}

public void outletOff() {
    requireActivity().runOnUiThread(() -> {
        binding.outlet2.setVisibility(View.INVISIBLE);
        binding.outlet.setVisibility(View.VISIBLE);
    });
}
```

```
case "/outlet/on":
    fragment.outletOn();
    break;
case "/outlet/off":
    fragment.outletOff();
    break;
```

برای روشن یا خاموش کردن پریز مطابق شکل سمت راست اینکه باید خاموش یا روشن شود را از uri استخراج کرده و مطابق شکل سمت چپ پریز خاموش یا روشن می‌شود.

پرده برقی:

```
public void curtainOpen() {
    requireActivity().runOnUiThread(() -> {
        binding.curtain.setVisibility(View.INVISIBLE);
        binding.curtain2.setVisibility(View.VISIBLE);
    });
}

public void curtainClose() {
    requireActivity().runOnUiThread(() -> {
        binding.curtain2.setVisibility(View.INVISIBLE);
        binding.curtain.setVisibility(View.VISIBLE);
    });
}
```

```
case "/curtain/open":
    fragment.curtainOpen();
    break;
case "/curtain/close":
    fragment.curtainClose();
    break;
```

برای باز یا بسته کردن پرده برقی مطابق شکل سمت راست اینکه باید باز یا بسته شود را از uri استخراج کرده و مطابق شکل سمت چپ پرده باز یا بسته می‌شود.

حسگر دما و حسگر روشنایی:

```
public int getTemp() { return temp; }

public int getLight() { return light; }
```

```
case "/temp":
    response += fragment.getTemp();
    break;
case "/light":
    response += fragment.getLight();
    break;
```

برای دریافت میزان دما و روشنایی مطابق شکل سمت راست دستور مورد نظر تشخیص داده شده و با استفاده از توابع شکل سمت چپ مقدار را به بدنه پاسخ اضافه می‌کند.

حسگر حرکت:

این حسگر کمی متفاوت از دو حسگر دیگر است، بدین صورت که هر زمان که دکمه تشخیص حرکت فشرده شد از طرف شبیه‌ساز باید درخواستی برای اطلاع رسانی تشخیص حرکت برای هاب فرستاده شود.

```
binding.moveButton.setOnClickListener(view -> {
    if (hubIp != null)
        client.newCall(new Request.Builder()
                .url(hubIp + "/move").build()).enqueue(new Callback() {
            @Override
            public void onFailure(@NonNull Call call, @NonNull IOException e) {
            }

            @Override
            public void onResponse(@NonNull Call call, @NonNull Response response) {
            }
        });
});
```

این درخواست با استفاده از یک client از کتابخانه OkHttp برای پورت 8080 همان آدرسی که درخواست conf/ را فرستاده بود، فرستاده می‌شود.