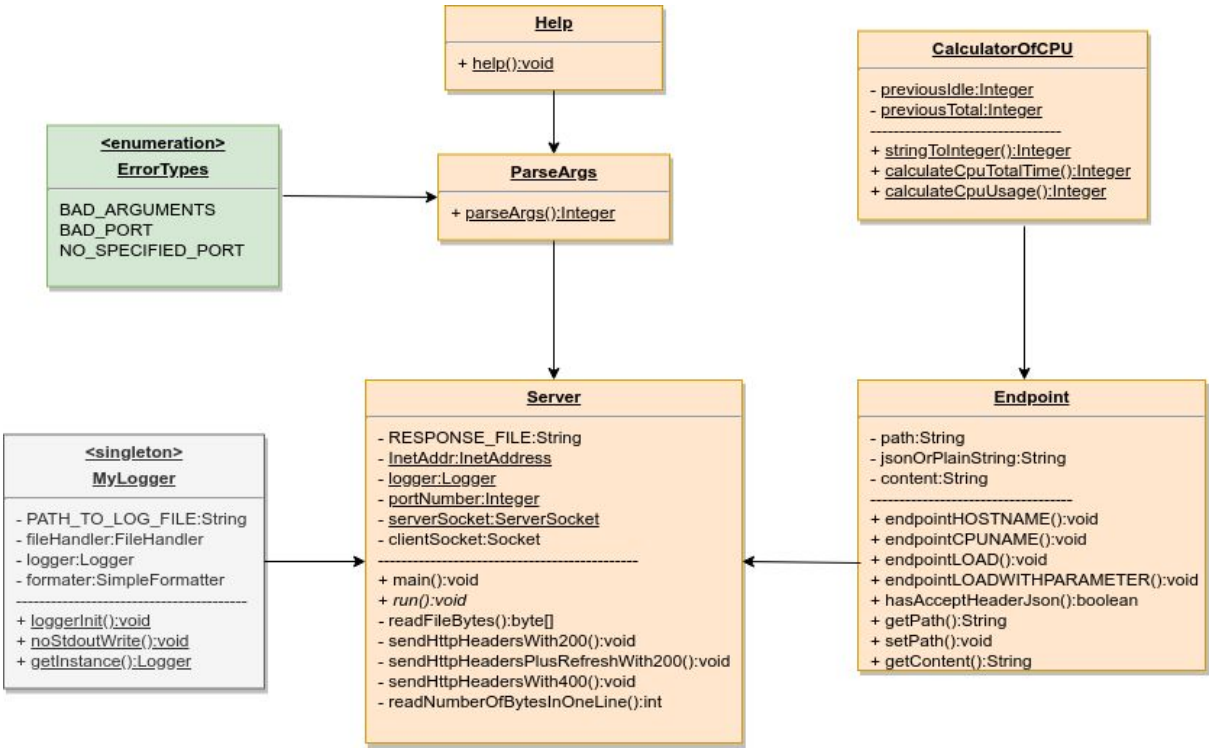
Server poskytující informace o systému pomocí HTTP

Návrh

Špecifikácia projektu bola celkom jasná a teda bolo pre mňa veľmi jednoduché daný server navrhnúť. Predtým ale ako som začal celý server navrhovať tak som si prečítal veľa podstatných informácií o aplikačnom protokole HTTP. Hlavným zdrojom informácií boli RFC. Na začiatku som si vytvoril jednoduchý class diagram, podľa ktorého som to následne aj implementoval. Ako môžete vidieť z **obrázku č.1**, tak som si daný projekt rozčlenil na viacero častí. Prvá časť ktorú som začal navrhovať boli triedy Server, Mylogger, Help a ParseArgs. Najpodstatnejšou už zo zmienených bola samozrejme Server. Druhou časťou bolo vytvorenie Endpoint triedy a nakoniec CalculatorOfCPU triedy. Jednotlivé závislosti môžete vidieť na diagrame.



obrázok č.1

Popis tried stručne:

- *Trieda help*
  - slúži iba ako pomoc pri zlých pokusov o spustenie aplikácie
- *Trieda ParseArgs*
  - kontrola, vstupných argumentov zadaných pomocou makefile
- *Enum ErrorTypes*
  - výčet chýb
- *Trieda Mylogger*
  - slúži na zachytenie histórie pri komunikácií so serverom
- *Trieda CalculatorOfCPU*
  - slúži na výpočet zaťaženia procesora
- *Trieda Endpoint*
  - slúži na spracovanie jednotlivých požiadavkou zaslaných klientmi
- *Trieda Server*
  - jadro celej aplikácie, vykonáva všetky časti čo boli spomenuté v daných triedach a plus sa stará o zaslanie jednotlivých http hlavičiek.

Implementácia

Ako som už zmienil v úvode tak som si pre implementáciu aplikácie zvolil práve známy jazyk Java. Pre zlepšenie motivácie a taktiež sumarizovanosť som mal vytvorené určité milníky aplikácie Každý milnik predstavoval daný celok, po ktorom som vždy validoval, zda sa aplikácia chová tak ako má.

V triede Server.java môžeme vidieť celú aktivitu, od request až po samotný response.

Všetko to začína v main metóde kde si inicializujeme logger, ktorý si na začiatku spustenia aplikácie vytvorí súbor log.0 a do tohto súboru bude zapisovať všetku komunikáciu, ktorá za daný čas nastala. Čo je ale taktiež vhodné spomenúť je, že ak daný súbor log.0 bude plný tzn. prekročí hranicu 16MB, hneď na to sa všetky aktivity zapísané na log.0 prepíšu do súboru log.1 a táto logika platí až po 5 súbor, kde sa nakoniec súbory prepisujú a máme tak v priemere viditeľnú aktivitu 2 - 4 dni v závislosti na zaťažení serveru.

Po vytvorení loggeru sa presúva do ďalšej časti a tou je práve spracovanie argumentov. V triede ParseArgs.java kontrolujeme spravnosť zadaného portu. Verifikácia je veľmi jednoduchá a mohla by sa určite zlepšiť. Mojou myšlienkou hlavne bolo zakázať danému uživateľovi zadanie portov 21, 22, 53, 80 a veľa ďalších obsadených pod hranicou 1000 čo sú porty pre FTP, SSH, DNS a HTTP.

V prípade ak sa podarí užívateľovi zadať adekvátny port, vytvára sa server socket, ktorý bude naslúchať na porte špecifikované od užívateľa a bude na adrese, ktorú si sám zvolí v našom prípade sa jedná o adresu serveru merlin.

Najpodstatnejšou časťou aplikácie je v metóde main cyklus while v ktorej daný server naslúcha a čaká na budúcich klientov. Každý klient predstavuje jedno vlákno. Zaručené je to

kvôli implementovania rozhrania Runnable. Je viacero spôsobov ako danú problematiku vyriešiť a to napríklad taktiež aj pomocou dedičnosti. Pomocou metódy .start() sa spustí metódu run() a teda náš klient. Na začiatku celého vykonávania sa vytvorí inštancia endpointu, ktorá je zodpovedná za všetky requesty poslané od klientov. Následná časť je zabalená do veľkého “try with resource” bloku podporované od JDK 8, pomocou ktorého umožňuje, že sa dané buffery zatvoria implicitne. Hlavičky zaslané od klienta sa spracovávajú v bufferi. Z prvého riadku sa zistí časť o aký endpoint sa bude jednať a akú metódu sa jedná. V našej aplikácie podporujeme iba metódu GET, čiže napríklad POST, PUT, DELETE nie sú prípustné. Dôležitá časť je pri selekcií jednotlivých endpointov, kde pomocou verifikácie pomocou regulárnych výrazov vyberie ten zvolený od klienta a následne sa vykoná požadovaná funkcionalita.

API poskytuje 4 typy endpointov:

1. /hostname
    - vráti sieťové meno počítača vrátane domény, napríklad:
2. cpu-name
    - vráti informáciu o procesore
3. /load
    - vráti aktuálne informácie o záťaži CPU
4. /load?refresh=X
    - vráti aktuálne informácie o záťaži CPU každých X sekúnd

Každý endpoint môže vrátiť ako typ dát jednoduchý text (text/plain) alebo známy typ používaný pre serializáciu json (application/json). Druh typu, ktorý bude poslaný klientovi záleží na požiadavke klient. Tým je myslené ak v hlavičkách HTTP je hlavička Accept s atribútom application/json data sú klientovi poslané vo forme jednoduché jsonu. Každý endpoint má rozdielnu implementáciu.

Endpoint:

1. hostname
    - hodnota získavaná z inštancie ServerSocket
2. cpu-name
    - data získavané pomocou príkazu lscpu
3. load
    - data získané zo súboru /proc/stat
    - využíva sa tu trieda CalculatorOfCPU
4. load?refresh=X
    - aplikovaná rovnaká logika ako z load ale plus využitie refresh hlavičky

Koncový stav serveru je vlastne po spracovaný endpointu a následne vyplnenie hlavičiek kde už na základe načítaných dát posiela odpoveď.

Súčasne server podporuje 3 typy stavových kódov:

1. 200 (OK)
    - indikuje, že žiadosť bola úspešná
2. 400 (BAD REQUEST)
    - označuje, že server nemôže alebo nebude žiadosť spracovávať z dôvodu, čo je vnímané ako chyba na strane klienta ako napríklad chybná syntax žiadosti
3. 405 (METHOD NOT ALLOWED)
    - indikuje, že metóda prijatá v žiadosti je známa serverom ale nie je podporovaná cieľovým zdrojom

Použitie

Aplikácia sa zapína a prekladá pomocou programu Makefile. Pred zapnutým serveru musíme samozrejme najskôr preložiť dané triedy. To jednoducho spravíme pomocou príkazu **make build.** Potom budeme môcť schopný jednoducho aplikáciu zapnúť ďalším príkazom **make run port=X**, kde X je port servera na ktorom bude naslúchať.

Príklad:

1. make build
2. make run port=1233

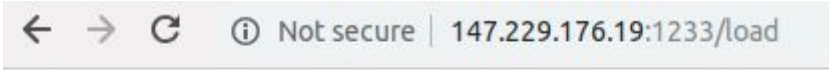
Server naslúcha na porte 1233.
Teraz môžeme jednoducho dotazovať náš server pomocou spomenutých endpointov:

1. pomocou webového prehliadača
2. pomocou programu curl
3. pomocou programu wget

1.Webový prehliadač


```
merlin.fit.vutbr.cz
```


```
Intel(R)Xeon(R)CPUE5-26400@2.50GHz
```




```
9%
```

2.curl

Pri použití curl programu je hlavne potrebné použiť prepínač -w ‘\n’, kedže curl implicitne odmazava ‘\n’ je potrebné explicitne toto pridať aby sme mali výstup presne zarovnaný výsledok. Taktiež si budeme môcť špecifikovať typ.

Získané data ako text/plain

```
majkl@majkl:~$ curl -w '\n' http://147.229.176.19:1233/cpu-name
Intel(R)Xeon(R)CPUE5-26400@2.50GHz
```

Získané data ako application/json

- pridaný -H prepínač na pridanie hlavičky accept s application/json

```
majkl@majkl:~$ curl -w '\n' http://147.229.176.19:1233/load -H 'Accept: application/json'
{"cpu-usage": "4%"}
majkl@majkl:~$ curl -w '\n' http://147.229.176.19:1233/load -H 'Accept: application/json'
{"cpu-usage": "7%"}
```

3.wget

```
majkl@majkl:~$ wget http://147.229.176.19:1233/hostname
--2019-02-13 21:03:39--  http://147.229.176.19:1233/hostname
Connecting to 147.229.176.19:1233... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19 [text/plain]
Saving to: ‘hostname’

hostname              100%[=============================>]      19  --.-KB/s    in 0s

2019-02-13 21:03:39 (65,9 KB/s) - ‘hostname’ saved [19/19]
```

Stiahnutý súbor, v ktorom už máme dáta vo forme text/plain **merlin.fit.vutbr.cz**

Rozšírenia a Záver

Medzi rozšírenia, ktoré som v projekte spravil bol Logger o ktorom som písal už v predošlých sekciách. Do budúcna by bolo možné veľmi jednoducho rozšíriť daný server už o ďalšie HTTP metody ako napríklad GET, PUT a pod.

Tento projekt hodnotím za veľmi prínosný z hľadiska informácií, ktoré som sa naučil o protokole HTTP. Vyskúšal som si spracovanie jednotlivých hlavičiek a ich následné dynamické modifikovanie. Bolo tam mnoho častí, ktoré si bolo nutné ozrejmiť a dopodrobna pochopiť.

Autor

- Maroš Orsák (xorsak02)