Metodika - Manažment kódu

Na *GitHube* existuje tímová organizácia tp17-2021. Tam sa nachádzajú všetky kódy potrebné pre samotné voľby ale aj pre stránku tímu a zvyšný projektový manažment. Všetci členovia tímu tam majú správcovský prístup a vedia teda upravovať a vytvárať repozitáre.

Pozor! Všetky repozitáre sú public, preto je potrebné dbať na to, aby sa v nich nenachádzali žiadne citlivé údaje ako napríklad API kľúče alebo prihlasovacie údaje.

Pozor! Niektoré existujúce repozitáre majú slovenské názvy, ale všetko nové je potrebné vytvárať a písať v angličtine.

Kedy je potrebný nový repozitár

Nezávislé komponenty aplikácie alebo služby je potrebné oddeliť do rôznych repozitárov. Nový repozitár je preto potrebné vytvoriť v prípade, že vyvíjaný komponent je nezávislý na ostatných. Pod nezávislým komponentom si je možné predstaviť časť aplikácie, ktorá je ohraničená špecifikovaným rozhraním (rozhranie v zmysle architektúry softvéru) a je možné ju považovať za jeden logický celok.

Pri *Microservice* architektúre by mala mať každá služba vlastný repozitár, pričom služby z rovnakého systému by mali mať rovnaký prefix v názve repozitáru. Napríklad *gateway-modules* a *gateway-token-manager*. Ďalej by pri Microservices mal existovať jeden repozitár napr. *gateway*, ktorý by obsahoval rôzne alternatívy *Docker compose* súborov pre rôzne situácie. Napr. *deploy*, *load_test*, *e2e_test*, *staging* a podobne.

Tuto je demo repoztiár MS projektu. Dá sa ním inšpirovať, ale nesedia k tejto metodike napríklad názvy repozitárov. Ale je možné si aspoň vytvoriť predstavu o štruktúre.

Pri delení kódu na komponenty je potrebné riadiť sa architektúrou aplikácie, prípadne sa poradiť so zvyškom tímu alebo architektom, ak je určený.

Ako sa vytvára repozitár

Na stránke organizácie na *GitHube* je potrebné v časti *Repositories* kliknúť na *New*.

- V kolonke *Owner* je potrebné vybrať organizáciu *tp17-2021*, nie osobný účet alebo inú svoju organizáciu.
- Do kolonky *Repository name* je potrebné zadať rozumný samoopisný názov nového repozitáru v angličtine oddelený pomlčkami, ktorý by mal hlavne zodpovedať danému komponentu alebo inému kódu, ktorý sa bude nachádzať v repozitári. Pri viacerých repozitároch týkajúcich sa nejakej spoločnej časti je vhodné zadať rovnaký prefix do názvu repozitáru. Napríklad *gateway-modules* a *gateway-token-manager*.
- Do *Description* je potrebné zadať v angličtine niekoľko slovný popis toho, čo sa bude nachádzať v repozitári.
- Viditeľnosť repozitáru je potrebné nastaviť na *Public*.
- V poslednom rade je potrebné, aby repozitár obsahoval *README.md* a *.gitignore* súbory. *README.md* je možné nastaviť už v tomto kroku zaškrtnutím checkboxu, ak súbory ešte žiadne *README.md* neexistuje.

Na druhej strane *.gitignore* je lepšie vytvoriť až manuálne počas vývoja, lebo ponúkaná možnosť vytvorenia na tejto obrazovke je tak trochu pofidérna.
- Nakoniec je potrebné kliknúť na *Create repository*.

Výsledkom tohto je vytvorený nový repozitár.

Ako si naklonovať repozitár

Na *Githube* je potrebné otvoriť daný repozitár. Vpravo hore sa nachádza zelené tlačidlo *Code*. Po jeho stlačení sa vyroluje menu pre klonovanie. Je odporúčané používať SSH možnosť. Je tam zobrazená adresa repozitára - text v tvare `git@github.com:...` Toto je potrebné si skopírovať. V konzole na svojom počítači je potrebné prejsť do adresára, kam bude repozitár naklonovaný. Repozitár naklonujeme príkazom git clone `git@github.com:...`

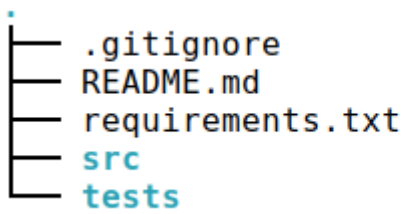
Poznámka: git clone vytvorí v aktuálnom adresári nový adresár s názvom podľa názvu repozitára. Často sa stáva, že človek si vytvorí adresár s takým názvom a očakáva, že git clone rovno do aktuálneho adresára stiahne už obsah repozitára. Nie, vytvorí ešte podadresár.

Ako si naklonovať repozitár, ak už lokálne prebehol git init

Rovnako ako v predošlom postupe, je potrebné si skopírovať si adresu repozitáru. V lokálnom repozitári je potrebné spustiť `git remote add origin git@github.com:...` Potom je potrebné commitnúť spraviť pull a vyriešiť prípadné konflikty.

Ideálnym riešením problémov tohto spôsobu je nepoužívať ho a najprv vytvoriť repozitár a až tak začať niečo kódiť lokálne.

Štruktúra repozitáru



Je odporúčané, aby repozitár napríklad Python API komponentu mal aspoň takú štruktúru ako na obrázku. V `requirements.txt` by mal byť zoznam potrebných knižníc. V `src/` by mal byť zdrojový kód a v `tests/` by mali byť testy.

Každá *Microservice* by mala mať aj *Dockerfile*.

Git účty v PC

Ak človek používa vo svojom PC rôzne *GitHub* účty, je potrebné si dať pozor na to, aký účet používa v konkrétnom repozitári. Globálne a lokálne nastavenia (globálne sú default pre celý PC, lokálne sú pre aktuálny repozitár) je možné si zobraziť príkazmi `git config -l` a `git config --local -l`. Pre úprave nezrovnalostí je odporúčané použiť editovací mód: `git config --local -e`.

Git a SSH

Niekedy sa stáva, že pri každom pulle a pushi musí vývojár zadávať svoje prihlasovacie údaje na *GitHub*. To je pomerne nepohodlné. Preto je odporúčané používať SSH kľúče. V nastavení svojho účtu na *github.com* v záložke *SSH and GPG keys* treba použiť *New SSH key*. Do *Title* sa zadáva vlastný názov kľúča a do *Key* je potrebné nakopírovať svoj public key. Na *Linuxe* a *Mac OS* je toto triviálna vec. Na *Windowse* sa často vyskytujú problémy a chce to viac námahy na sfunkčnenie. Používateľom *Windowsu* ostáva indviduálne si vygoogliť, ako to rozbehať alebo najlepšie použiť tento návod.

README.md

Každý repozitár by mal obsahovať *README.md* v angličtine s vysvetlením, čo sa nachádza v repozitári, prípadne, ako to spustiť a ako to funguje.

Branche

Po novom (niekoľko mesiacov) sa hlavná brancha na *GitHube* volá *main*, nie *master* ako v minulosti. Hlavnej branchi je potrebné nastaviť zákaz priameho pushovania a yvžadovanie akceptovaného pull requestu pred mergovaním. To je možné nastaviť v nastaveniach repozitára -> *Branches* -> *Add rule*. Do *Branch name pattern* je potrebné zadať názov branche, teda *main*. V *Protect matching branches*, je potrebné vybrať *Require a pull request before merging* a v tom *Require approvals*. Samozrejme, podľa potreby je možné nastavovať ďalšie obmedzenia na ďalších branchiach.

Do *main* by sa malo mergovať iba ak je mergovaný kód spustiteľný a bez chýb (nemusí byť ešte úplný). Napríklad, sú špecifikované funkčné požiadavky komponentu, takže po každej plne a korektne naimplementovanej funkcionalite je možné mergnúť vývojové branche do *mainu*.

Pre vývoj je potrebné používať vývojové branche. Každý repozitár by mal mať jednu *development* branchu a potom niekoľko ďalších pre individuálne potreby. Napríklad, jedna brancha pre každú funkcionalitu. To už je na racionálnom zvážení vývojára v danom repozitári.

Čo sa týka názvu branche, je potrebné dodržiavať spojovníkový formát (foo-bar-aha). Okrem samotného identifikátora Jira tasku, ktorý má v sebe pomlčku.

Ak sa brancha týka konkrétneho tasku v Jire, je vhodné zahrnúť identifikátor tasku v názve branche. Najlepšie ako prefix. Napríklad, *EV-123-million-dollar-problem-sol*.

Commit message

Asociácia s Jira taskami

Ak sa daný commit týka nejakého tasku, je nutné v commit message uviesť identifikátor tohto tasku. Každú prácu je možné namapovať na nejaký task, takže je očakávané, že každý jeden commit bude asociovaný s nejakým taskom pomocou identifikátora v commit message. Príklady commit messages: *EV-72 Test new github pipeline*, *EV-89 Update webpage assets - add python code style guide*.

Informatívne popisy

V commit message je potrebné rozumne opísať, čo daný commit prináša, čo mení a podobne. Všetko je potrebné písať v angličtine.

Commit message by nemala presiahnuť dĺžku 72 znakov.

Od štvrtákov na FIIT STU je očakávané, že vedia písať rozumné commit message bez toho, aby dostali papekom po hlave.

Pull requesty

Pull request (PR) je v podstate požiadavka na merge nejakej branche do nejakej inej. Teda, ak je kód pripravený na mergnutie do nadradenej branche, vývojár vytvorí PR, lebo chce mergnúť tento nový kód do nadradenej branche.

Vytvorenie

Na *GitHube* nájde tlačidlo *New pull request* a stalčí ho. Alternatívne sa nachádza na branchi, ktoré chce mergnúť, a *GitHub* mu už zobrazuje možnosť vytvorenia PR.

PR je potrebné rozumne nazvať. Defaultne tam *GitHub* dá commit message posledného commitu mergovanej brenche. Avšak, výpovednejšie je pri viacerých commitoch zhrnúť v názve podstatu PR. Netreba ale presahovať cca 5 slov. Ak sa PR týka konkrétneho tasku, je vhodné ho referencovať identifikátorom v názve PR.

Ďalej je potrebné pridať komentár/popis k PR. Tento môže byť ľubovoľne dlhý. Tu je vhodné zhrnúť, čo má vývojár na srdci a prečo si myslí, že by mal byť tento PR odobrený a mergnutý do nadradenej branche. Zoznam commitov tu nie je potrebné písať, pretože to pri PR vidno aj tak.

Ak je repozitár nastavený podľa tejto metodiky, pre prijatie PR je potrebná aspoň jedna review. Pri vytváraní PR je možné v pravej časti obrazovky pridať reviewerov. Ak je dôvod (napr. je to reporter toho tasku alebo je to človek, ktorý má na starosti túto časť, alebo existuje hocijaký iný dôvod, prečo by tam mal byť práve on) na to, aby tam bol niekto konkrétny, je potrebné ho pridať.

Okrem toho je možné pridať PR nejaké labele. Defaultne tam je niekoľko fajných. Avšak, oplatí sa vždy si vytvoriť ešte jeden “bugfix” napríklad defaultne žltou farbou. Totiž, často sa stane (aj keď by sa nemalo), že až po mergnutí je objavená chyby a je potrebné ju rýchlo opraviť a znova mergnúť. Vtedy je PR označený napr. týmto labelom.

Následne je potrebné kliknúť *Create pull request* alebo *Create draft pull request*, ak je známe, že kód ešte nebude možné prijať, ale je potrebné vytvoriť PR, aby sa rozbehla diskusia a vyriešil sa v nej nejaký konkrétny problém.

Diskusia

Pri PR je možné písať komentáre a viesť tak diskusiu ohľadom prinesených zmien, otázok, objavených chýb a podobne. Tu si väčšinou vývojári vyjasňujú veci okolo daného PR a navrhujú zmeny.

Ak chce vývojár povoliť PR alebo formálne požiadať o zmeny, je potrebné, aby sa najprv pridal medzi reviewerov daného PR -> vpravo hore -> *Reviewers* -> klikne na seba. Potom je potrebné obnoviť stránku. Hore na stránke sa zobrazuje v žltom boxe oznam o tom, že PR čaká na jeho review. Vývojár potrebuje stlačiť zelené tlačidlo *Add your review*.

Na nasledujúcej obrazovke vidí diffy zmenených súborov. Vpravo hore je tlačidlo *Review changes*. Po kliknutí na toto tlačidlo vie vývojár napísať nejaký komentár a vybrať si, či je to iba komentár, alebo *Approve* alebo formálne *Request changes*.

Vývojár sa môže zapojiť do diskusie normálne aj keď nie je formálne medzi reviewermi, ale vtedy nevie udeliť approve na merge.

Uzavretie

Ak je už všetko ok, reviewer PR by mal po *Review changes* stlačiť Approve. Následne už vie ľubovoľný vývojár stlačiť *Merge pull request* a *Confirm merge*.

CI/CD pipelines

Na *GitHube* je možné pekne si automatizovať deployovanie a tiež testovanie. Volá sa to *Actions*. Tie je možné nastaviť tak, aby s aautomaticky spúšťali pri určitých eventoch v codebase. Napríklad, spustiť automaticky deploy stránky po prijatí pull requestu alebo automaticky spustiť testy kódu po vytvorení pull requestu a teda pred potenciálnym mergnutím, aby bolo hneď jasné, či kód prejde testami.

Tieto *Actions* je možné spúšťať na *GitHub* serveroch ale aj na vlastných. Náš projekt používa self-hosted workera na *team17-21.studenti.fiit.stuba.sk* serveri.

Automatické deployovanie už zrejme inde ako na tímovej stránke nebude potrebné použiť. Ak by sa aj naskytla príležitosť, je možné to vyriešiť individuálne. Je ale veľmi žiadané použiť automatické testy v projekte.

Testovanie

GitHub má na internete takýto celkom pekný návod k testovaniu Pythonu v actions. Ale v skratke:

```yaml
# nejaký názov pipeline
name: Run tests on XY branch

# kedy sa má automaticky spúšťať
on:
  # pri každom push do branche development
  push: [ development ]

jobs:
  build:

    # kde sa to má spúšťať (pre tento projekt je nakonfigurovaná self-hosted group)
    runs-on: self-hosted

    # čo sa má vykonať
    steps:
      # naklonuje aktuálnu branchu do work directory: /home/ubuntu/actions-runner/_work/...
      - uses: actions/checkout@v2

      # názov kroku
      - name: echo hello world
        # ľubovoľný linux command na runs-on^ stroji v tom work directory
        run: echo "Hello world!"

      # nejak tak sa vraj testuje
      - name: Test with pytest
        run: |
          pytest
```

Toto je *GitHub* pipeline. V repozitári je to súbor: `test/.github/workflows/názov.yml`. Na obrázku je vysvetlená základná syntax a funkcionalita.

V prípade Microservice architektúry s viacerými repozitármi je pre komplexnejšie testovanie aplikácie potrebné v pipeline naklonovať viaceré potrebné repozitáre. Ak ešte používame aj Docker, v pipeline je potrebné postaviť image, potom spustiť kontajner a otestovať požadovaný interface.

Dockerfile

`Dockerfile` je špeciálny súbor, ktorý predpisuje *Dockeru*, ako má vyskladať požadovaný image z danej aplikácie. Príklad `Dockerfilu` pre vytvorenie *FastAPI* imagu:

```
# predpripravený image z dockerhubu
FROM python:3.10

# hlavný priečinok v kontajneri
WORKDIR /code

# nakopírovanie súboru requirements.txt do kontajnera
COPY ./requirements.txt ./code/requirements.txt

# inštalácia potrebných knižníc podľa requirements.txt
RUN pip install --no-cache-dir --upgrade -r ./code/requirements.txt

# nakopírovanie zdrojových kódov do kontajnera
COPY ./src /code/src

# sputenie FastAPI služby na porte 80
CMD [ "uvicorn", "src.main:app", "--host", "0.0.0.0", "--port", "80" ]
```

Jednotlivé kroky je možné vnímať ako vrstvy pridávané do imagu/kontajnera (kontajner je konkrétna inštancia imagu). Ak by bolo pri každom jednom spustení potrebné sťahovať a inštalovať potrebné súbory, build time by bol nepohodlne vysoký. Preto si *Docker* chachuje image po vrstvách s tým, že ak sa obsah vrstvy nezmenil, použije sa nacacheovaná verzia a pokračuje sa ďalším krokom. Od prvej zmenenej vrstvy je ale nutné odznova sťahovať, kopírovať a inštalovať aj všetky ďalšie vrstvy. Z tohto dôvodu je pre optimalizáciu build time vhodné zoradiť kroky v `Dockerfile` podľa frekvencie zmien v nich od najmenšej.

Konkrétne v tomto príklade sú najprv kopírované `requirements.txt` a sú nainštalované a až tak sú kopírované zdrojové kódy aplikácie. Je očakávané, že zdrojové kódy sa budú meniť možno aj pri každom builde, no potrebné knižnice sa zmenia iba výnimočne. Preto ich stačí v skutočnosti nainštalovať iba raz, čo zebrie netriviálny čas, a po zvyšok vývoja *Docker* používa nacacheovanú verziu týchto nainštalovaných knižníc, čo zaberá prakticky 0 času.

Docker build

Pre vytvorenie imagu z `Dockerfile` je potrebné zavolať príkaz `docker build`. Príklad:

```
docker build -t image-name .
```

Toto vytvorí image podľa `Dockerfile` v aktuálnom priečinku a názve ho `image-name`. (tá bodka na konci značí aktuálny priečinok, nie je možné ju vynechať)

Docker run

```
docker run -d --name container-name -p 8222:80 image-name
```

Toto vytvorí a spustí inštanciu imagu `image-name` v kontajneri `container-name`.

Prepínač `-d` zabezpečí spustenie v detached mode, čiže nezostane v termináli otvorené spojenie do vnútra kontajneru, ale spustí sa na pozadí.

Prepínač `-p 8222:80` nabinduje port 80 vo vnútri kontajnera na port 8222 lokálneho počítača.

Docker-compose

V prípade potreby zhlukovania viacerých kontajnerov (ako napríklad pri Microservices) je vhodné použiť docker-compose. Ide o predpis toho, aké kontajnery je potrebné spustiť a aké parametre im nastaviť. Dobrým príkladom tohto je *gateway* repozitár, ktorý zhlukuje viacerá služby G a spája ich jedným `docker-compose.yml` súborom. Okrem toho je v tomto repozitáry ilustrované použitie *GitHub submodules*, teda referencovanie rôznych repozitárov v jednom hlavnom.

Týmto príkazom je možné skomponovať a rovno spustiť kontajnery na pozadí:

```
docker-compose up -d
```

Užitočné docker príkazy

```
docker ps -a
```

Zobrazí všetky existujúce kontajnery.

```
docker logs container-name
```

Vypíše logy daného kontajnera.

```
docker stop container-name
```

Zastaví kontajner (ten bude stále existovať, ale bude zastavený).

```
docker start container-name
```

Rozbehne kontajner, ak je zastavený.

```
docker rm container-name
```

Vymaže kontajner. Najprv musí byť zastavený.