SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Tím 16 - MI16

Inteligentný importér verejných datasetov

Metodika pre vývoj a prehliadku zdrojového kódu

Vedúci tímu:
Ing. Jakub Šimko, PhD.

Členovia tímu:
Bc. Ladislav Bari
Bc. Adam Ševčík
Bc. Adam Talian
Bc. Filip Varga
Bc. Milan Vaško
Bc. Jana Vrabľová

Dedikácia

Tento dokument predstavuje metodiku pre vývoj a prehliadku zdrojového kódu na tímovom projekte tímu 16 (MI-16) na FIIT STU v akademickom roku 2017/2018. Metodika je určená pre všetkých členov tímu.

Postup

Všetok napísaný kód súvisí s úlohami, ktoré sú vytvorené v nástroji JIRA. Životný cyklus týchto úloh je popísaný v metodike manažmentu úloh. Tento dokument bližšie rozoberá stavy Ready to Review, In Review a Ready to Deploy.

1. Keď je programátor, ktorý pracoval na úlohe, spokojný s vykonanou prácou a skontroloval jej funkčnosť, presunie svoju úlohu do stavu Ready to Review. Zároveň s týmto presunom musí programátor v nástroji Gitlab otvoriť Merge Request - požiadavku na merge do vetvy development.
2. Kontrolovač si stiahne zdrojový kód z vetvy, ktorú programátor použil na vývoj danej funkcionality.
3. Kontrolovač krátko skontroluje, či dodaný kód funguje, ako má.
4. Následne robí prehliadku kódu na základe pravidiel písania zdrojového kódu, ktoré sú uvedené nižšie.
5. V prípade, že kontrolovač nájde niečo, čo nevyhovuje popísanej kvalite, pomocou nástroja Gitlab nechá autorovi Merge Requestu komentár, na ktorý musí programátor reagovať. Výsledkom je diskusia, ktorá končí jednou z dvoch možností:
    a. V prípade, že programátor má pádne argumenty vysvetľujúce a zdôvodňujúce diskutovaný kód, kontrolovač uzavrie diskusiu bez ďalších zmien.
    b. Ak je kód programátora naozaj chybný, musí programátor pripraviť nový commit, ktorý adresuje pripomienky kontrolovača. Ten následne tento commit zhodnotí opäť - výsledkom je buď ďalšie pokračovanie diskusie, alebo jej uzavretie.
6. Ak sú všetky diskusie uzavreté, kontrolovač urobí jednu z nasledujúcich akcií:
    a. Ak kód vyhovuje štandardom kvality, vetva s kódom sa mergne do vetvy development podľa metodiky pre verziovanie.
    b. Ak kód nevyhovuje štandardom kvality, Merge Request je zamietnutý bez vykonania mergu.

Pravidlá písania zdrojového kódu

Vyvíjaný projekt je aktuálne rozdelený na 2 časti:

- Frontend písaný v jazyku TypeScript s využitím rámca Angular
- Backend písaný v jazyku Java s využitím rámca Spring Boot

Táto sekcia je v dôsledku tejto skutočnosti tiež rozdelená na 2 časti - každá rozoberá jednu časť projektu.

Prvé pravidlo, ktoré je dôležitejšie než všetky ostatné, je: **Používať common sense**. Ak niečo dáva zmysel, funguje to a je to elegantné, má to prioritu pred týmto dokumentom.

TypeScript - Angular

Používať pravidlá definované súborom tslint.json - množstvo nájstrojov poskytuje zvýrazňovanie chýb na základe pravidiel definovaných v tomto súbore.

Typy

Treba využívať typy čo najviac, ako sa len dá. Slabinou jazyka TypeScript je, že predstavuje nadstavbu nad JavaScriptom, takže je jednoduché zabudnúť uviesť typ - treba si na to dávať pozor. Kód bez typov je síce kratší, ale nezachytí takmer žiadne chyby.

Zle (posielanie HTTP požiadaviek):

```
public saveAttribute(model, token) {
    const result = {
        uri: '',
        label: model.label,
        owlCategory: model.owlCategory,
        group: '',
        description: model.description,
        similarPhrase: Array.isArray(model.similarPhrase) ? model.similarPhrase :
[model.similarPhrase],
        flag: true,
    };

    return this.http.post(`${AppConfig.SERVER}/addAttribute`, result, {
        headers: this.getHeaders(),
        params: { token },
    });
```

```
}
```

Dobre (posielanie HTTP požiadaviek):

```
public saveAttribute(model: OWLDataObject, token: string): Observable<OWLDataObject> {
    const result: OWLDataObject = {
        uri: '',
        label: model.label,
        owlCategory: model.owlCategory,
        group: '',
        description: model.description,
        similarPhrase: Array.isArray(model.similarPhrase) ? model.similarPhrase :
[model.similarPhrase],
        flag: true,
    };

    return this.http.post<OWLDataObject>(`${AppConfig.SERVER}/addAttribute`, result, {
        headers: this.getHeaders(),
        params: { token },
    });
}
```

Zle (prijímanie HTTP požiadaviek):

```
this.findAttributeService.find(this.searchQuery).subscribe(
    (values) => {
        this.possibilities = values;
        this.paginator.pages = Math.ceil(values.length / this.paginator.perPage);
    },
    (error) => {
        this.notification.showErrorTranslatedHttp(
            'attributeSearch.error.search.title',
            'attributeSearch.error.search.message',
            error
        );
    },
);
```

Dobre (prijímanie HTTP požiadaviek):

```
this.findAttributeService.find(this.searchQuery).subscribe(
    (values: OWLDataObject[]) => {
        this.possibilities = values;
        this.paginator.pages = Math.ceil(values.length / this.paginator.perPage);
```

```
  },
  (error: HttpErrorResponse) => {
    this.notification.showErrorTranslatedHttp(
      'attributeSearch.error.search.title',
      'attributeSearch.error.search.message',
      error
    );
  },
);
```

Organizácia súborov

Všetky súbory súvisiace s komponentom by mali ísť spolu do jednej zložky (služby (service), testy, pomocné triedy...).

Oddelene (v zdieľanej zložke) by sa mali nachádzať len:

- naozaj zdieľané komponenty (autocomplete, dropdown, spinner...)
- model pre sieťovú komunikáciu s backendom

HTTP komunikácia (a chybové stavy všeobecne)

Vždy je potrebné ošetriť chybový stav a zobraziť používateľovi informáciu o tom, že sa niečo pokazilo (s čo najpresnejším popisom).

Zle:

```
this.findAttributeService.find(this.searchQuery).subscribe(
  (values: OWLDataObject[]) => {
    this.possibilities = values;
    this.paginator.pages = Math.ceil(values.length / this.paginator.perPage);
  },
);
```

Dobre:

```
this.findAttributeService.find(this.searchQuery).subscribe(
  (values: OWLDataObject[]) => {
    this.possibilities = values;
    this.paginator.pages = Math.ceil(values.length / this.paginator.perPage);
  },
  (error: HttpErrorResponse) => {
    this.notification.showErrorTranslatedHttp(
      'attributeSearch.error.search.title',
      'attributeSearch.error.search.message',
      error
```

```
        );
    },
);
```

Nemanipulovať s DOM priamo (jQuery a podobne)

V starom frontende (pred refaktorom) bolo na veľa miestach používané jQuery na ručnú manipuláciu s DOM. Angular bol stvorený práve kvôli tomu, aby toto už nebolo potrebné. Na strane TypeScriptu sa musí riešiť logika, nie dizajn!

Zle (zo starého frontendu):

```
    // Vyskytuje sa tu veľa logiky súvisiacej s rozložením stránky.
    // Nič z toho nie je potrebné!
    // Vždy je potrebné riešiť dizajn pomocou HTML a SCSS, do TypeScriptu
    // treba zasahovať len v prípade krajnej núdze (tento príklad v krajnej
    // núdzi nebol, po vymazaní dizajnovej logiky z TypeScriptu sa stránka
    // dokonca zobrazovala lepšie).

    public moveTable(value: number): void {
        this.maxOffset = $("#tableHead").width() - $(".pane-hScroll").width();
        this.actualOffset += value * EditComponent.SPEED;
        this.actualOffset = Math.max(0, this.actualOffset);
        $(".pane-hScroll").scrollLeft(this.actualOffset);
    }

    public ngAfterViewInit(): void {
        this.actualOffset = 0;
        this.maxOffset = $("#tableHead").width() - $(".pane-hScroll").width();
        $("th").each(function(index) {
            const th = $(this);
            const width = th.find(".d-inline-block").width();
            $("td:nth-child(" + (index + 1) + ")").width(width);
            th.width(width);
        });
        this.cdr.detectChanges();
    }

    public ngOnInit(): void {
        this.entityService.getEntities().subscribe(
            (response: Entity[]) => this.entities = response,
            (error) => this.notificationService.showErrorMessage(error));

        // this.entities = this.entityService.fake_getEntities();
```

```
        this.maxOffset = $("#tableHead").width() - $(".pane-hScroll").width();
        this.cdr.detectChanges();

            ...
    }
```

Dobre (čo zo starého kódu zostalo):

```
    public ngOnInit(): void {
        this.entityService.getEntities().subscribe(
            (response: ResponseObject<EntitiesContent>) => this.entities =
response.content.entities,
            (error: HttpErrorResponse) => {
                this.notification.showErrorTranslatedHttp('edit.error.init.title', 'edit.error.init.message',
error);
            },
        );
    }
```

Využívať niektoré princípy funkcionálneho programovania (map, filter...)

Možno trochu kontroverznejší bod metodiky, ale najmä v situáciách, kedy sa zoznam jedného typu transformuje na iný zoznam obsahujúci iný typ hodnôt sa kód väčšinou zjednoduší pri použití metódy map (v kombinácii s inými funkciami, napríklad filter - článok pre inšpiráciu: https://medium.com/poka-techblog/simplify-your-javascript-use-map-reduce-and-filter-bd02c593cc2d)

Zle:

```
this.columnData = []
for (const row of this.fileData.uniqueData) {
    const el = row[this.columnIndex];
    if (el !== null) {
        this.columnData.push(el);
    }
}
```

Dobre:

```
this.columnData = this.fileData.uniqueData
    .map((row) => row[this.columnIndex])
    .filter((el) => el !== null);
```

Java - Spring Boot

Formát

Všetok kód je formátovaný defaultným formatterom v IntelliJ. Toto formátovanie sa dá spustiť klávesovou skratkou Ctrl + Shift + L.

Try catch

Vo väčšine prípadov treba pri ošetrení výnimky preferovať deklaráciu throws pred try - catch blokom.

Zle:

```
public String doStuff(String path) {
    try {
        return readFile(path);
    } catch (IOException e) {
        logger.error(e);
    }
}
```

Dobre:

```
public String doStuff(String path) throws IOException {
    return readFile(path);
}
```

Ak sa jedná o chybu, ktorá je očakávaná a program po nej môže pokračovať, môže byť vhodnejšie použiť aj try - catch:

```
public int getWithDefault(String key, int defaultValue) {
    try {
        return storage.getValue(key);
    } catch (KeyNotFoundException e) {
        return defaultValue;
    }
}
```

Uzatváranie zdrojov (resource) a try-catch with resources

Ak zatvárame nejaký zdroj dát (napr. File), treba použiť try-catch with resources:

Zle:

```
public void saveFile(long id) throws IOException {
   File file = openFile(id);
   processFile(file);
   file.close();
}
```

Dobre:

```
public void saveFile(long id) throws IOException {
   try (File file = openFile(id)) {
      processFile(file);
   }
}
```

Životnosť premenných

Všetky premenné by mali mať najkratšiu možnú životnosť a mali by byť deklarované čo najbližšie ich použitiu:

Zle:

```
public ResponseObject getResponseObject() {
   ResponseObject responseObject = new ResponseObject();

   FileContent fileContent = new FileContent();
   fileContent.setData(fileService.getData());

   responseObject.setContent(fileContent);
   return responseObject;
}
```

Dobre:

```
public ResponseObject getResponseObject() {
   FileContent fileContent = new FileContent();
   fileContent.setData(fileService.getData());

   ResponseObject responseObject = new ResponseObject();
   responseObject.setContent(fileContent);
   return responseObject;
}
```

Zbytočný stav

Tiež je preferované nezneužívať členské premenné na účel posúvania argumentov. Členské premenné sú potrebné na uchovávanie stavu objektu - ak nie je uchovávanie stavu potrebné, je lepšie sa mu vyhnúť.

Zle:

```
public class FileReader {
   private String path;

   public FileReader(String path) {
      this.path = path;
   }

   public byte[] read() {
      File file = new File(path);
      return IOUtil.readBytes(file);
   }
}
public class Main {
   public static void main(String[] args) {
      FileReader reader = new FileReader("/path/to/file.txt");
      byte[] data = reader.read();
      ...
   }
}
```

Lepšie:

```
public class FileReader {
   public static byte[] read(String path) {
      File file = new File(path);
      return IOUtil.readBytes(file);
   }
}
public class Main {
   public static void main(String[] args) {
      byte[] data = FileReader.read("/path/to/file.txt");
      ...
   }
}
```

Spring Boot spôsob:

Tento spôsob je preferovaný.

```
@Service
public class FileReader {
    public byte[] read(String path) {
        File file = new File(path);
        return IOUtil.readBytes(file);
    }
}
@Controller
public class SomeController {
    @Autowired
    private FileReader fileReader;

    public void someFunction() {
        byte[] data = fileReader.read("/path/to/file.txt");
        ...
    }
}
```