Arhitektura i projektovanje softvera
Model komunkacije

Strahinja Laktović 15691
Julije Kostov 15680

Tipovi komunikacije:
- Komunikacija izmedju mikroservisa
- Komunikacija izmedju komponenata sistema

Tehnologije:
- Moleculer framework
- Pusher

Moleculer

Prednosti mikroservisa u NodeJS-u:
- Aplikacija se pokreće brže, što čini developere produktivnijim i ubrzava deployment.
- Lakše je skalirati razvoj i ima prednosti u performansama.
- Eliminiše dugoročnu predanost nekom steku tehnologija. Razvojem novog servisa može se izabrati novi stek.
- Svaki servis može biti objavljen nezavisno od drugih - lakši i češći deployment novih verzija servisa.

Kao framework zadužen za komunikaciju mikroservisa korišćena je Moleculer Service broker komponenta.

Svaki od mikroservisa definiše svog brokera i akcije koje drugi servisi mogu pozvati korišćenjem svog, koristeći RMI način komunikacije.
Broker može definisati evente, kao i handlere za svoje ali i evente drugih brokera u sistemu, čime se implementira publish/subscribe tip komunikacije izmedju mikroservisa.

Broker - Javascript implementacija

```
const { ServiceBroker } = require('moleculer');
const { redis } = require('config');
const events = require('./events');
const actions = require('./actions');

const initBroker = () => {
  const broker = new ServiceBroker({
    nodeID: 'redis-micro-1',
    logger: process.env.NODE_ENV === 'development',
    logLevel: 'info',
    transporter: {
      type: 'Redis',
      options: {
        host: redis.host,
      },
    },
  });

  broker.createService({
    name: 'medicine',
    events,
    actions,
  });

  return broker;
};

module.exports = initBroker;
```

```
const logger = require('services/logger');
const drugService = require('services/redis/drug');
const { getSideEffectsByDrugName } = require('api/drug');

module.exports = {
  'drug.prescribed': async (data) => {
    try {
        const { drugName } = data;
        const { data: sideEffects } = await getSideEffectsByDrugName(drugName);

        const se = await drugService.getSideEffectsForDrug(drugName);

        if(se.length === 0) {
            await drugService.setSideEffectsForDrug(drugName, sideEffects);
        }
    } catch (ex) {
      logger.error(ex);
    }
  },
};
```

```
module.exports = {
  async getDiseaseByName(ctx) {
    const { name } = ctx.params;

    try {
        const disease = await diseaseService.getDisease(name);

        if(Object.keys(disease).length) {
            return { disease };
        }

        const { data } = await getDiseaseByName(name);

        diseaseService.setDisease(name, data.disease);

        return data;
    } catch (e) {
        return null;
    }
  },

  async getSideEffectsByDrugName(ctx) {
    const { drugName } = ctx.params;

    try {
      const sideEffects = await drugService.getSideEffectsForDrug(drugName);

      if(sideEffects.length) {
          return sideEffects;
      }

      const { data } = await getSideEffectsByDrugName(drugName);

      drugService.setSideEffectsForDrug(drugName, data);

      return data;
  } catch (e) {
      return null;
  }
},

  async getDiseasesNameLike(ctx) {
    const { name } = ctx.params;

    try {
      const { data } = await getDiseasesNameLike(name);
      return data;
    } catch (e) {
      return null;
    }
  },

  async getMedicationsForDisease(ctx) {
    const { name } = ctx.params;

    try {
      const medications = await diseaseService.getDrugsForDisease(name);

      if(medications.length) {
          return medications;
      }

      const { data } = await getDrugsForDisease(name);

      await diseaseService.setDrugsForDisease(name, data);

      return data;
    } catch (e) {
      return null;
    }
  },
};
```

Broker - Typescript implementacija

```typescript
import {ServiceBroker} from 'moleculer';

export class Broker {
    private static broker: Broker = new Broker();
    private readonly serviceBroker: ServiceBroker;
    private constructor() {
        this.serviceBroker = new ServiceBroker( options: {
            nodeID: 'api-1',
            logger: true,
            logLevel: 'info',
            transporter: {
                type: 'Redis',
                options: {
                    host: 'redis',
                },
            }});
    }

    public static getInstance() {
        return Broker.broker.serviceBroker;
    }

    public static startBroker() {
        Broker.getInstance().start();
    }

    public static async getDiseasesNameLike(name) {
        return await Broker.getInstance()
            .call( actionName: 'medicine.getDiseasesNameLike',  params: { name: name });
    }

    public static async getMedicationsForDisease(name) {
        return await Broker.getInstance()
            .call( actionName: 'medicine.getMedicationsForDisease',  params: { name:name });
    }

    public static async getDisease(name) {
        return await Broker.getInstance().call( actionName: 'medicine.getDiseaseByName',  params: { name });
    }

    public static async getSideEffectsForDrug(drugName) {
        return await Broker.getInstance().call( actionName: 'medicine.getSideEffectsByDrugName',  params: { drugName });
    }

    public static emitPrescribedEvent(drugName) {
        Broker.getInstance().emit( eventName: 'drug.prescribed',  payload: { drugName });
    }

}
```

Pusher

Za komunikaciju izmedju komponenti sistema korišćen je Pusher, tačnije Pusher Channels komponenta.

Pusher Channels obezbedjuje komunikaciju u realnom vremenu izmedju servera, aplikacija, uredjaja i ima široku primenu.
Pusher Channels implementira publish/subscribe model.

Primer korišćenja Pushera

Stranica pacijenta web aplikacije zainteresovana za promene merenja nekog pacijenta može da se subscribuje na kanal koji u imenu sadrži id tog pacijenta. Dodavanjem novog merenja iz mobilne aplikacije Pusher servis implementiran na backendu publishuje novokreirani objekat. Stranica web aplikacije u realnom vremenu prikazuje promene na grafikonima koristeći Pusher servis i dispatchovanjem asinhrone Redux akcije.

```
import { Injectable} from '@nestjs/common';
import { IPusherService} from "./interfaces/pusher-service.interface";
import * as Pusher from "pusher";
import {Measurement} from "../measurements/entity/measurement.entity";
import {EventType} from "./enum/event-type.enum";
import {User} from "../users/entity/user.entity";
import {Prescription} from "../prescriptions/entity/prescription.entity";
import {Examination} from "../examinations/entity/examination.entity";
import {ConfigService} from "../config/config.service";
import {IPusherOptions} from "./interfaces/pusher-options.interface";

@Injectable()
export class PusherService implements IPusherService {
    constructor(private readonly configService: ConfigService ) {
        this.pusher = new Pusher( opts: { ...this._options });
    }

    private readonly pusher: Pusher;

    private _options: IPusherOptions = {
        appId: this.configService.get('PUSHER_APP_ID'),
        key: this.configService.get('PUSHER_KEY'),
        secret: this.configService.get('PUSHER_SECRET'),
        cluster: this.configService.get('PUSHER_CLUSTER'),
        useTLS: true,
    };

    public async createMeasurement(measurement: Measurement, userId): Promise<void> {
       this.pusher
            .trigger(
                 channel: `measurements-${userId}`,
                EventType.Create,
                 data: { measurement }
            );
    }

    public async updateUser(user: User): Promise<void> {
        this.pusher
            .trigger(
                 channel: `users-${user.id}`,
                EventType.Update,
                 data: { user }
            );
    }

    public async createPrescription(prescription: Prescription, userId): Promise<void> {
        this.pusher
            .trigger(
                 channel: `prescriptions-${userId}`,
                EventType.Create,
                 data: { prescription }
            );
    }

    public async createExamination(examination: Examination, userId): Promise<void> {
        this.pusher
            .trigger(
                 channel: `examinations-${userId}`,
                EventType.Create,
                 data: { examination }
            );
    }
}
```

```
componentDidMount() {
  const {
    getPatientAction,
    match:{ params: { id } },
    getChartsAction,
    addMeasurementPushAction,
    updatePatientPushAction,
  } = this.props;

  getChartsAction(id);
  getPatientAction(id);

  pusherService.subscribe(`measurements-${id}`, eventType.Create, addMeasurementPushAction);
  pusherService.subscribe(`users-${id}`, eventType.Update, updatePatientPushAction);
}

componentWillUnmount() {
  const {
    match:{ params: { id } },
  } = this.props;
  pusherService.unsubscribe(`measurements-${id}`);
  pusherService.unsubscribe(`users-${id}`);
}
```

```
import * as Pusher from "pusher-js";
import config from './config';

class PusherService {
  constructor() {
    const { key, additionalConfig } = config;
    this.pusher = new Pusher(key, additionalConfig);
  }

  subscribe(channel, event, callback) {
    this.pusher
      .subscribe(channel)
      .bind(event, ({ data }) => callback(data));
  }

  unsubscribe(channel) {
    this.pusher.unsubscribe(channel);
  }
}

const pusherService = new PusherService();

export default pusherService; //singleton
```