# Web aplikacija za automatizirano prikupljanje sadržaja s YouTube usluge

**Getto, Vjekoslav**





**SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU**

**FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA**

**Sveučilišni studij**

# WEB APLIKACIJA ZA AUTOMATIZIRANO PRIKUPLJANJE SADRŽAJA S YOUTUBE USLUGE

**Diplomski rad**

**Vjekoslav Getto**

**Osijek, 2017.**

# SADRŽAJ

# 1. UVOD

Moderne web aplikacije napravljene pomoću JavaScript skriptnog jezika omogućavaju programerima jednostavnu i brzu izradu korisničkog i poslužiteljskog koda pomoću istog programskog jezika. Odabir *JavaScript-a* kao programskog jezika, programeru omogućava korištenje velikog broja gotovih biblioteka i pruža mogućnost dijeljenja koda između klijentske i poslužiteljske aplikacije.

Zadatak rada je napraviti web aplikaciju čiji će se sadržaj automatski prikupljati s *YouTube* servisa u obliku kodova koji su dio putanja (engl. *link, URL*) do videa. Aplikacija se sastoji od administratorskog i korisničkog sučelja. Administratorsko sučelje omogućava administratoru upravljanje i objavljivanje sadržaja koji će biti prikazan u korisničkom sučelju, dok će korisničko sučelje sadržavati pregled i pretragu sadržaja prema ključnim riječima (engl. *tags*).

Rad se sastoji od tri dijela: poslužiteljski dio, klijentski dio i struktura aplikacije. U prvom dijelu rada opisan je poslužiteljski (engl. *backend*) dio aplikacije, modeli i baza podataka te prikupljanje i pohrana sadržaja. U drugom dijelu opisan je klijentski (engl. *frontend*) dio aplikacije u kojoj je objašnjen administratorski i korisnički dio aplikacije. Posljednja cjelina objašnjava korištenje aplikacije uz slikovni pregled.

Tijekom izrade rada kao primjer sadržaja korišten je *YouTube* kanal koji se bavi taktikom i komentiranjem MMA i boks borbi.

## 1.1. Zadatak završnog rada

Zadatak završnog rada je napraviti web aplikaciju, pomoću *JavaScript* skriptnog jezika, čiji je poslužiteljski dio zadužen za pružanje sadržaja klijentskom dijelu aplikacije te za prikupljanje sadržaja s *YouTube* servisa.

Klijentski dio aplikacije sastoji se od administratorskog i korisničkog sučelja. Korisničko sučelje je namijenjeno korisnicima koji će moći pretraživati i pregledavati prikupljeni i objavljeni sadržaj.

Pomoću administratorskog sučelja, administratoru stranice omogućeno je pokretanje procesa prikupljanja sadržaja tako da unese poveznicu na određeni *YouTube* kanal, proizvoljno ime za taj *YouTube* kanal, te jednu ili više ključnih riječi (u daljnjem tekstu *tag*).

Administratorsko sučelje također treba omogućiti uređivanje *tag-ova*, dijeljenje pojedinog videa i pripadajućeg teksta na društvenoj mreži *Twitter*, te objavljivanje ili sakrivanje pojedinog videa na korisničkoj stranici.

# 2. POSLUŽITELJSKI DIO APLIKACIJE

Poslužiteljski dio aplikacije zadužen je za pružanje podataka za prikaz klijentskom dijelu aplikacije, upravljanje modelima te prikupljanje i pohranu sadržaja.

## 2.1. Korištene tehnologije i biblioteke

*NodeJS* je *run-time* okruženje koje uz *npm* (*Node Package Manager*) stvara moćan alat za programiranje poslužitelja. *Express.js* [4] je razvojna cjelina (engl. *framework*) za izradu web aplikacija, pomoću koje je napravljeno aplikacijsko programsko sučelje *(engl. API)* koje se koristi za komunikaciju između klijenta i poslužitelja. Uz *Express.js* razvojnu cjelinu koristi se i *body-parser* [11] biblioteka koja služi za obradu HTTP zahtjeva.

Za pokretanje poslužitelja koristi se *Nodemon* [5] uslužni program koji prati promjenu datoteka poslužitelja, te u slučaju promjene ponovno pokreće poslužitelj.

Za pohranu podataka se koristi *NoSQL* baza podataka *MongoDB* [6], te *mongoose* [7] biblioteka za modeliranje podataka, tj. stvaranje i upravljanje objektima u bazi podataka.

*PhantomJS* [8] memorijski internetski preglednik (engl. *headless web browser)* i *CasperJS* [9] biblioteka se koriste za učitavanje *YouTube* stranice u memoriju, prikupljanje sadržaja sa stranice i interakciju sa stranicom.

*Cron* [10] biblioteka koristi se za periodično izvršavanje poslova, npr. prikupljanja podataka s *YouTube-a* i praćenje svih korištenih *tag-ova* u bazi podataka.

Za dijeljenje sadržaja na popularnoj društvenoj mreži *Twitter* koristi se biblioteka *Twit* [12].

## 2.2. Modeli

### 2.2.1. Author

*Author* je osnovni model koji se koristi za prikupljanje podataka, pri čemu je pravilo da će poslužitelj pri svakoj iteraciji prikupljanja podataka pokušati pronaći novi video sadržaj za svaki spremljeni *Author* objekt. Svaki *Author* objekt sadržava sljedeća svojstva: ime autora

(*name*), poveznicu na *YouTube* kanal od autora (*url*), listu osnovnih *tag-ova* (*default_tags*), datum zadnje provjere podataka na *YouTube* servisu (*last_sync_date*), te identifikator (*last_url_code*), koji se koristi pri prikupljanju podataka. Uz ova svojstva, *MongoDB* baza podataka za svaki objekt kreira *id* svojstvo kao unikatnu referencu na taj objekt.

Shema modela je prikazana u programskoj implementaciji 2.1. .

```
var mongoose = require("mongoose"),
    Schema = mongoose.Schema;

var authorSchema = Schema({
    name: String,
    url: String,
    default_tags: [],
    last_url_code: String,
    last_sync_date: {
        type: Date,
        default: Date.now
    }
});
module.exports = mongoose.model("Author", authorSchema);
```

**Programska implementacija 2.1.** Shema Author modela (sheme, engl. Scheme)

### 2.2.2. YoutubeLink

Programska implementacija 2.2. prikazuje shemu modela *YoutubeLink* koji sadrži podatke o svakom prikupljenom videu. *YoutubeLink* model sadrži svojstvo tipa *Schema.ObjectId*, koja označava da se radi o referenci na drugi model, u ovom slučaju *Author* model (svojstvo *_author_id)*. Ostala svojstva ovog modela su: naslov videa (*title*), kod koji je dio poveznice na *YouTube* video (*url*), duljina videa (*duration*), lista ključnih riječi (*tags)* te trenutačno stanje (*state*).

```
var youtubeLinkSchema = Schema({

    _author_id: Schema.ObjectId,

    title: String,

    url: String,

    duration: String,

    tags: [],

    state: String

});
```

**Programska implementacija 2.2.** Shema YoutubeLink modela

### 2.2.3. Tag

*Tag* je model koji se koristi za praćenje svih *tag-ova* koji se pojavljuju na stranici i njihov broj ponavljanja. Model sadrži svojstvo *text* gdje se nalazi tekst *tag-a* i svojstvo broja ponavljanja *count*. Svojstvo *count* odgovara broju aktivinih *YouTubeLink* objekata u bazi podataka, koji sadrže taj *tag*. (Programska implementacija 2.3.)

```
var tagSchema = Schema({

    text: String,

    count: Number,

});
```

**Programska implementacija 2.3.** Shema Tag modela

### 2.2.4. TagCronQueue

*TagCronQueue* model sastoji se od dvije liste, novih (*newTags*) i starih (*oldTags*) *tag-ova*, a shema modela prikazana je u programskoj implementaciji 2.4. *TagCronQueue* objekti se koriste pri obavljanju *Cron* posla koji je zadužen za održavanje *count* svojstava *Tag* objekata.

```
var tagCronQueueSchema = Schema({

    newTags: [],

    oldTags: [],

});
```

**Programska implementacija 2.4.** Shema TagCronQueue modela

## 2.3. Poslužitelj i kontroleri

Glavna datoteka poslužitelja je *server.js*, a ona se pokreće pomoću uslužnog programa *Nodemon* (Programska implementacija 2.5.*),* koji prati promjene u datotekama poslužitelja, te na svaku promjenu datoteka ponovno pokreće poslužitelj, te čini isto u slučaju pogreške poslužitelja.

```
nodemon server.js
```

**Programska implementacija 2.5.** Pokretanje poslužitelja

Poslužiteljev zadatak je stvoriti *Express.js* objekt, preko kojega se definiraju putanje i kontroleri za svaku putanju, a pomoću kojih klijentska aplikacija komunicira s poslužiteljem. Nakon definiranja putanja, poslužitelj se pokreće pozivom funkcije *listen*, te je dostupan preko porta 7777. (Programska implementacija 2.6.).

```
var express = require("express");

var bodyParser = require("body-parser");

var path = require("path");

var adminAuthorController = require("./controllers/admin/authorController");

var adminLinkController = require("./controllers/admin/adminLinkController");

var adminTwitterController = require("./controllers/admin/adminTwitterController");

var linkController = require("./controllers/linkController");

var tagController = require("./controllers/tagController");


var app = express();

app.use(express.static(path.join(__dirname, "../app/dist")));

app.use(bodyParser.json());
```

```
app.use("/api/admin/authors", adminAuthorController);

app.use("/api/admin/youtubeLinks", adminLinkController);

app.use("/api/admin/twitter", adminTwitterController);

app.use("/api/links", linkController);

app.use("/api/tags", tagController);

app.listen(7777, function() {

    console.log("Started listening on port", 7777);

});
```

**Programska implementacija 2.6.** Definiranje putanja i kontrolera

Poslužitelj također kreira objekt biblioteke *mongoose,* preko kojeg se spaja na *MongoDB* bazu podataka, koju kontroleri koriste za pohranu podataka. Pri pokretanju poslužitelja, definiraju se i pokreću dva *Cron* posla (Programska implementacija 2.7*.). YoutubeFetchingCronJob* i *TagsCronJob,* čija će uloga biti detaljnije objašnjena u daljnjem tekstu.

```
var mongoose = require("mongoose");

var YoutubeFetchingCronJob = require("./YoutubeFetchingCronJob");

var TagsCronJob = require("./TagsCronJob");


mongoose.connect("mongodb://localhost/fightbreakdowns");

YoutubeFetchingCronJob.getCronJob().start();

TagsCronJob.getCronJob().start();
```

**Programska implementacija 2.7.** Spajanje na bazu podataka i pokretanje Cron poslova

### 2.3.1. Link Controller

*Link Controller* je zadužen za dostavljanje podataka o prikupljenim *YoutubeLink* objektima klijentskom dijelu aplikacije, te se koristi samo u klijentskom dijelu aplikacije koji je dostupan krajnjem korisniku. Postoje dvije putanje preko kojih se može pristupati *YoutubeLink* objektima.

Prva putanja je *"/front",* koja prihvaća HTTP GET metodu, a vraća sve *YouTubeLink* objekte iz baze podataka koji imaju svojstvo *state* jednako "*active"*, što znači da ova putanja vraća samo objekte koje je administrator stranice odabrao da budu dostupni krajnjem korisniku.

Druga putanja, *"/search"* koristi se za pretragu po *tag-ovima.* Potrebno je poslati HTTP POST [12] poziv koji sadrži *tag-ove* koje traženi *YouTubeLink* objekt treba sadržavati. Za pretragu se koristi *find* funkcija na *YoutubeLink* objektu, a kao argumenti pretrage koriste se *state* svojstvo koje treba biti "*active",* te posebno svojstvo *mongoose* biblioteke *$all*, koje označava da traženi objekt može sadržavati jedan ili više *tag-ova* koji se nalaze u nizu primljenom od klijenta prilikom POST poziva (*req.body.tags*). Programska implementacija ovog kontrolera prikazana je u programskoj implementaciji 2.8.

```
var router = require("express").Router();

router.route("/front").get(getLinks);

router.route("/search").post(searchLinks);


function getLinks(req, res) {

    YoutubeLink.find().where('state').equals("active").exec(function(err, links) {

        if (err)

            res.send(err);

        else {

            res.json(links);

        }

    });

}

function searchLinks(req, res) {

    YoutubeLink.find({

        tags: {

            "$all": req.body.tags

        },

        state: 'active'

    }).exec(function(err, links) {

        if (err)

            res.send(err);
```

```
        else {
            console.log(links);
            res.json(links);
        }
    });
}
```

**Programska implementacija 2.8.** LinkController implementacija

### 2.3.2. Tag Controller

*Tag Controller* se koristi za pristupanje svim pohranjenim *Tag* objektima, pomoću HTTP GET [12] poziva. Na *Tag* objektu poziva se funkcija *find* kojoj se prosljeđuje argument "*$gte: Number(1)"*, kako bi bili pretraživani samo *Tag* objekti koji imaju svojstvo *count* veće ili jednako broju jedan.

```
var router = require("express").Router();
router.route("/").get(getTags);

function getTags(req, res) {
    Tag.find({
        count: {
            $gte: Number(1)
        }
    }, function(err, tags) {
        if (err) {
            res.send(err);
        } else {
            res.json(tags);
        }
    });
}
```

**Programska implementacija 2.9.** TagController implementacija

### 2.3.3. Admin Link Controller

*Admin Link Controller* putanje dostupne su samo administratoru stranice, a kontroler se koristi za upravljanje *YouTubeLink* objektima. Putanje *"/fetched", "/active" i "inactive"* dostupne su preko HTTP GET metode, te vraćaju sve *YouTubeLink* objekte koji imaju svojstvo *state* jednako "*fetched", "active"* ili "*inactive",* ovisno o putanji.

Putanja *"/:authorId"* vraća sve *YouTubeLink* objekte čije je svojstvo *authorId* jednako onom proslijeđenom od klijenta HTTP GET metodom.

Za uređivanje *YouTubeLink* objekata koristi se *"/:youtubeLinkId"* putanja i HTTP PUT [12] metoda. Prilikom poziva na tu putanju i nakon uređivanja određenog objekta, kreira se *TagCronQueue* objekt kojega kasnije procesira *Cron* posao zadužen za upravljanje svojstvima *Tag* objekata. Djelomična programska implementacija ovog kontrolera prikazan je u programskoj implementaciji 2.10..

```
var router = require("express").Router();

router.route("/").get(getAllYoutubeLinks);

router.route("/fetched").get(getFetchedYoutubeLinks);

router.route("/active").get(getActiveYoutubeLinks);

router.route("/inactive").get(getInactiveYoutubeLinks);

router.route("/:authorId").get(getYoutubeLinksByAuthorId);

router.route("/:youtubeLinkId").put(putYoutubeLink);


function putYoutubeLink(req, res) {

    var lowerCaseTagsNoHash = req.body.tags.map(function(tag, index) {

        return tag.toLowerCase();

    });

    YoutubeLink.findById(req.body._id, function(err, youtubeLink) {

      var oldTags = youtubeLink.tags;

      youtubeLink.tags = lowerCaseTagsNoHash;

      youtubeLink.state = req.body.state;

      youtubeLink.save(function (err, link) {

        if (err) {

            res.send(err);
```

```
        } else {
            res.json(link);
            var tagCronQueue = new TagCronQueue({newTags: lowerCaseTagsNoHash, oldTags:
oldTags});
            tagCronQueue.save(function(error) {
              if (error) {
                console.log("Error when saving tagCronQueue document");
              }
            });
        }
      });
    });
}
```

**Programska implementacija 2.10.** AdminLinkController implementacija

### 2.3.4. Admin Twitter Controller

*Admin Twitter Controller* (Programska implementacija 2.11.) koristi se za objavljivanje sadržaja na društvenoj mreži Twitter uz pomoć bibilioteke *Twit.* Uz prethodno kreiran *Twitter* račun, potrebno je generirati ključeve i tokene kako bi se aplikaciji omogućilo upravljanje *Twitter* računom. Nakon kreiranja objekta *Twit*, na tom objektu poziva se *post* funkcija s argumentom *"statuses/updates"* i prosljeđuje se status poslan od klijentskog dijela aplikacije HTTP POST metodom (*req.body.tweet).*

```
var Twit = require("twit");
var T = new Twit({
  consumer_key:         "jsHyX32fsEFXFC2LpBfUQ9ZDC",
  consumer_secret:      "633OvVapckcfdffQWhloRdfj8mxBr7IKwV1Okh8SVbSLdLbBel",
  access_token:         "8729078268sdsf63552-CFIHI839KIjYCn6bvYhk9sKsypPLGoO",
  access_token_secret:  "nXWzuhxtssdgf3y1FW0jd9t6ZVc6CuYdMI6k08xG9Rgx9R",
  timeout_ms:           60*1000,
});
```

```
var router = require("express").Router();

router.route("/").post(postTwitterStatus);


function postTwitterStatus(req, res) {

    T.post('statuses/update', { status: req.body.tweet }, function(err, data, response) {

      if (err) {

        res.send(err);

      } else {

        res.json(response);

      }

    });

}
```

**Programska implementacija 2.11.** AdminTwitterController implementacija

### 2.3.5. Admin Author Controller

*Admin Author Controller* koristi se za dohvaćanje i kreiranje *Author* objekata. Za kreiranje *Author* objekata koristi se HTTP POST metoda, koja uz spremanje podataka u bazu podataka *Author* objekata, također kreira *TagCronQueue* objekt kojega procesuira *Cron* posao.

```
var router = require("express").Router();

router.route("/:id?").get(getAuthors).post(addAuthor);


function addAuthor(req, res) {

    req.body.default_tags = _string.words(req.body.default_tags);

    var author = new Author(_.extend({}, req.body));

    author.save(function(err) {

        if (err) {

          res.send(err);

        }

        else {

          res.json(author);
```

```
          var tagCronQueue = new TagCronQueue({newTags: req.body.default_tags, oldTags:
[]});

          tagCronQueue.save(function(error) {

            if (error) {

              console.log("Error when saving tagCronQueue document");

            }

          });

        }

    });

}
```

**Programska implementacija 2.12.** AdminAuthorController implementacija

## 2.4. Prikupljanje sadržaja s YouTube servisa

*PhantomJS* je *JavaScript* memorijski internetski preglednik koji nema korisničko sučelje, a *web* stranice se učitavaju u memoriju (engl. *headless browser). CasperJS* je biblioteka koju je moguće koristiti uz *PhantomJS,* koja služi za upravljanje i interakciju sa stranicom koja je učitana u memoriju.

Osnovna ideja prikupljanja sadržaja je otvoriti željeni *YouTube* kanal, te pomoću *CasperJS* funkcija pritiskati tipku *"Load More",* dok se ne učita sav video sadržaj tog kanala, koji već nije prikupljen u prethodnoj iteraciji prikupljanja.

Elementi *web* stranice potrebni za interakciju se pronalaze pomoću *XPath* putanja, a na isti način se pronalaze i željeni podatci o svakom *video-u.*

Skripta za prikupljanje sadržaja je strukturirana tako da prima kao argument putanju (*url)* do određenog *YouTube* kanala i proizvoljni argument s identifikatorom koji je zadnji prikupljeni *video (link)*, kako se ne bi preuzimali već preuzeti *linkovi.* Identifikator je pohranjen u *Author* objektu pod svojstvom *last_url_code.*

*Casper* objekt kreira se sa svojstvom *pageSettings* tako da internetski preglednik bude predstavljen *YouTube* servisu kao mobilni telefon, kako bi bio učitan jednostavniji oblik

stranice. Također je postavljeno svojstvo *loadImages* na "*false*", kako se ne bi učitavale slike, te je na taj način ubrzano učitavanje stranice.

Nakon pokretanja *CasperJS-a* s funkcijom *start,* otvara se željeni *URL* s *thenOpen* funkcijom, te se čeka učitavanje stranice s *waitFor* funkcijom. Nakon prvotnog učitavanja, poziva se funkcija *clickMore* koja je rekurzivna funkcija. (Programska implementacija 2.13.)

```
var casper = require('casper').create({

    logLevel: "debug",

    pageSettings: {

        userAgent: "Mozilla/5.0 (Linux; Android 4.1.1; Galaxy Nexus Build/JRO03C)
AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.166 Mobile Safari/535.19",

        "loadImages": false

    }

});

casper.start();

casper.thenOpen(url).then(function() {

    this.waitFor(function check() {

        return this.evaluate(fetchMobileYtLinkNumber) != 0;

    }, function then() {

        clickMore.call(casper);

    });

}).then(function() {

    captureAndWriteLog.call(casper);

}).run(function() {

    //console.log("done");

    this.exit();

});
```

**Programska implementacija 2.13.** Pokretanje skripte za prikupljanje sadržaja

Funkcija *clickMore,* u slučaju da je prikazana tipka *"Load More",* na učitanoj stranici poziva ponovno *clickMore* funkciju rekurzivno, sve dok se ne učitaju svi dostupni video sadržaji tog kanala. U slučaju kada *"Load More"* tipka više nije vidljiva, poziva se *fetchMobileYtLinks*

funkcija, koja preko *XPath* putanje pronalazi sve učitane video sadržaje na stranici, te vraća listu objekata koji sadržavaju naslov, trajanje i identifikator . Opisane funkcije prikazane su u programskoj implemetaciji 2.14.

```
function fetchMobileYtLinks() {
    var links = __utils__.getElementsByXPath("//a[contains(@href, '/watch?v=')]");
    var returnList = [];

    links.forEach(function(link) {
        var url = link.getAttribute('href');
        var title =
link.children[0].children[1].children[0].children[0].children[0].innerText;
        var duration = link.children[0].children[0].children[1].children[0].innerText;
        returnList.push({
            title: title,
            url: url.split('/watch?v=')[1],
            duration: duration
        });
    });
    return returnList;
}

function clickMore(max, i) {
    i = i || 0;
    if ((max == null || max == undefined || i < max) && this.visible(moreButton)) {
        var links = this.evaluate(fetchMobileYtLinks);
        updateLinkList(this.evaluate(fetchMobileYtLinks));

        if (!no_new_links) {
            this.thenClick(moreButton);
            this.waitUntilVisible(loadingButton);
            this.waitUntilVisible(moreButton, function then() {
                clickMore.call(this, max, i + 1); // recursion
            }, function onTimeout() {
```

```
                updateLinkList(this.evaluate(fetchMobileYtLinks));
            });
        }
    } else if (!this.visible(moreButton)) {
        updateLinkList(this.evaluate(fetchMobileYtLinks));
    } else {
        this.capture("Error.png");
    }
}
```

**Programska implementacija 2.14.** Funkcije za interakciju sa stranicom i prikupljanje sadržaja

## 2.5. Youtube Cron Job

*YoutubeFetchingCronJob* je *Cron* posao koji se izvršava svakih 30 minuta. Zadatak ovog posla je pozivati skriptu za prikupljanje podataka za svaki *Author* objekt u bazi podataka. Skripta se poziva pomoću *Exec* biblioteke koja omogućava izvršavanje naredbi u terminalu.

Skripta za prikupljanje podataka s *YouTube* servisa, objašnjena u prošlom poglavlju, kao rezultat vraća listu novih podataka u *JSON* formatu. Vraćeni podaci se obrađuju te spremaju u bazu podataka. Djelomična programska implementacija ovog posla prikazana je u programskoj implementaciji 2.15. .

```
    function executeFetcherForNextAuthor() {
        var author = authorList.shift();
        if (author === undefined) {
            isRunning = false;
            return;
        }
        var execString = "casperjs link_fetcher.js " + author.name.replace(/ /g, '') + " "
+ author.url + " " + author.last_url_code;
        exec(execString, function(err, stdout, stderr) {
            try {
                var ytLinks = JSON.parse(stdout);
```

```
ytLinks.forEach(function(link) {
    var youtubeLink = new YoutubeLink({
        _author_id: author._id,
        _author_name: author.name,
        title: link.title,
        url: link.url,
        tags: author.default_tags,
        duration: link.duration,
        state: "fetched"
    });

    youtubeLink.save(function(err) {
        if (err) {
            console.log("Error when saving youtube link.");
            isRunning = false;
        }
    });
});

if (ytLinks.length > 0) {
    author.last_url_code = ytLinks[0].url;
} else {
    console.log("No new links from " + author.name);
}

author.last_sync_date = Date.now();

author.save(function(err) {
    if (err) {
        console.log("Error when saving author" + err);

    }
});
```

```
            } catch (e) {
                console.log(e);
                isRunning = true;
            }
            executeFetcherForNextAuthor();
        });
    }

    function getCronJob() {
        return new cronJob('*/30 * * * * *', function() {
            if (!isRunning) {
                isRunning = true;
                Author.find(function(err, authors) {
                    if (err) {
                        console.log("Error when fetching authors..");
                        isRunning = false;
                    } else {
                        authorList = authors;
                        executeFetcherForNextAuthor();
                    }
                });
            } else {
                console.log("Fetcher already running, aborting..");
            }
        }, null, true, 'America/Los_Angeles');
    }    return {
        getCronJob: getCronJob
    }
```

**Programska implementacija 2.15.** Djelomična implementacija YouTubeFetchingCronJob-a

# 3. KLIJENTSKI DIO APLIKACIJE

U ovom poglavlju opisane su korištene tehnologije i arhitektura pri izradi korisničkog sučelja. Arhitektura korisničkog sučelja objašnjena je primjerom kroz komponentu za pretragu video sadržaja.

## 3.1. Korištene tehnologije

Za izradu korisničkog sučelja korištena je poznata *JavaScript* biblioteka *React* [13].

Izrada korisničkog sučelja s *React* bibliotekom temelji se na konceptu komponenti (engl. *Component),* od kojih svaka ima svoje unutarnje stanje (engl. *state).* Prilikom izrade komponenti ideja je kreiranje manjih komponenti koje zajedno tvore jednu veću komponentu, tj. cijelu stranicu. *React* sučelja kreiraju se pomoću skriptnog jezika *JSX* [14], koji je zapravo *JavaScript* jezik s dodatkom *XML* standarda.

Važno svojstvo *React* biblioteke je virtualni *Document Object Model* (engl. *Virtual DOM)* koji omogućava da internetski preglednik na promjenu stanja određene komponente ponovno generira samo nužni *HTML kod,* a ne HTML kod cijele stranice. Svrha virtualnog DOM-a je ubrzati generiranje promjena u korisničkom sučelju.

Osim *React* biblioteke, koristi se i *Gulp* [17] alat za automatiziranije poslova (engl. *Task Runner),* te *browserify* [15], *reactify* [16] *i vinyl-source-stream* alati, kako bi se napisani programski kod generirao u oblik koji internetski preglednik može učitati.

## 3.2. Gulp, browserify i reactify

Programska implementacija 3.1. prikazuje dva *Gulp* posla za korisnički dio aplikacije, koji se zovu *bundle* i *copy*. Uz navedene poslove, postoje i dva identična posla za administratorski dio aplikacije (*bundleAdmin i bundleCopy).*

*Bundle* posao, učitava datoteku *main.jsx,* te skriptni*JSX* kod iz datoteke pretvara u *JavaScript* kod pomoću *browserify* i *reactify* alata, jer internetski preglednici ne mogu procesuirati *JSX* programski kod. Nakon pretvorbe u *JavaScript* kod, pomoću *vinyl-source-stream* alata pretvoreni kod se upisuje u *main.js* datoteku u *"app/dist"* direktorij. Nakon što je završen

*bundle* posao, potrebno je pokrenuti *copy* posao, koji kopira datoteke *index.html, bootstrap.min.css* i *style.css,* također u "*app/dist"* direktorij.

Nakon što su ova dva posla obavljena, u *"app/dist"* direktoriju su dostupne sve potrebne datoteke koje *Node.js* poslužitelj može posluživati internetskom pregledniku.

```
var gulp = require("gulp");

var browserify = require("browserify");

var reactify = require("reactify");

var source = require("vinyl-source-stream");


gulp.task("bundle", function () {

    return browserify({

        entries: "./app/main.jsx",

        debug: true

    }).transform(reactify)

        .bundle()

        .pipe(source("main.js"))

        .pipe(gulp.dest("app/dist"))

});


gulp.task("copy", ["bundle"], function () {

    return gulp.src(["app/index.html","app/lib/bootstrap-
css/css/bootstrap.min.css","app/style.css"])

        .pipe(gulp.dest("app/dist"));

});
```

**Programska implementacija 3.1.** Gulp poslovi za korisnički dio aplikacije

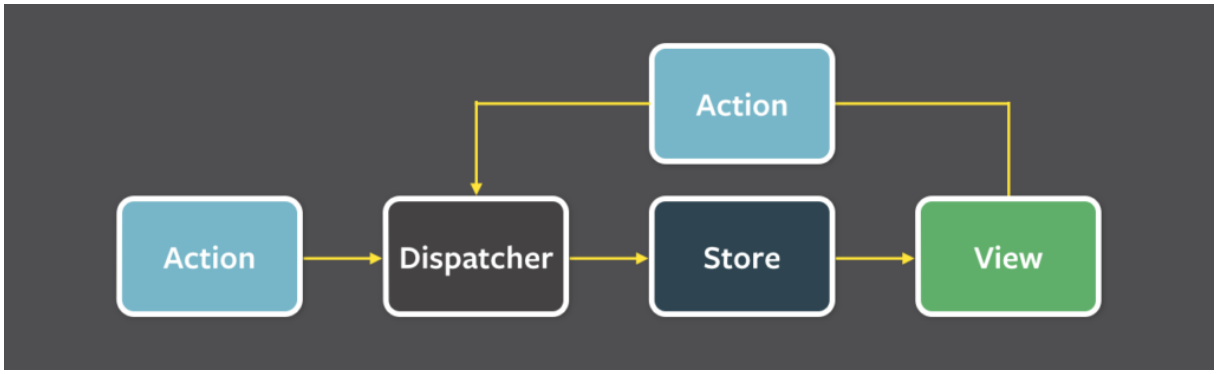
## 3.3. Flux arhitektura

*Flux* arhitektura je jedna od preporučenih arhitektura pri izradi *React* korisničkih sučelja. Zadatak *Flux* arhitekture je programeru ponuditi smjernice pri izradi korisničkog sučelja kako

bi tok podataka bio jednosmjeran, te se na taj način postiže preglednost samog koda kao i olakšano održavanje i izmjenjivanje programskog koda kroz vrijeme.

*Flux* arhitektura podjeljena je na četiri komponente: akcije (engl. *actions),* dispečer (engl. *dispatcher)*, spremišta (engl. *stores),* prikaz (engl. *views).*

Tijek podataka u *Flux* arhitekturi prikazan je na slici 3.1.




**Slika 3.1.** Tijek podataka u Flux arhitekturi

*Flux* arhitektura će biti objašnjena na primjeru komponente za pretraživanje.

### 3.3.1 Dispatcher

Programska implementacija 3.2. prikazuje programski kod *dispatcher* komponente. *Dispatcher* komponenta sadrži dvije funkcije, *register* i *dispatch. Register* metoda kao argument prima funkciju, te se ta funkcija sprema u niz pod nazivom *listeners*, pod unikatnim ključem koji se dobije pomoću *Guid* objekta.

Metoda *dispatch* prima kao argument objekt, koji će potom biti odaslan svim funkcijama koje su registrirane preko *register* funkcije i spremljene u *listeners* niz.

```
var Guid = require("guid");

var listeners = [];


function dispatch(payload) {
```

```
    for (var id in listeners) {
        listeners[id](payload);
    }
}
function register(cb) {
    var id = Guid.create();
    listeners[id] = cb;
}
module.exports = {
    register: register,
    dispatch: dispatch
}
```

**Programska implementacija 3.2.** Dispatcher

### 3.3.2. Actions

Akcije se pozivaju nakon interakcije korisnika s aplikacijom. Zadatak akcija je pozvati *dispatch* funkcije na *dispatcher* objektu, preko koje se prosljeđuju podaci o korisnikovoj interakciji. Programska implementacija 3.3. prikazuje *LinkActions* komponentu. Nakon što korisnik pokrene pretragu po upisanim *tag-ovima*, poziva se *searchByTags* funkcija kojoj se prosljeđuje niz *tag-ova* koje je korisnik upisao. Potom se poziva *dispatch* funkcija kojoj se prosljeđuje objekt koji ima svojstvo *tags*, gdje se nalaze upisani *tag-ovi*, te svojstvo *type*, prema kojemu određena *store* komponenta zna da je objekt namijenjen njoj.

```
var dispatcher = require("../dispatcher");
module.exports = {
    searchByTags: function(tags) {
        dispatcher.dispatch({
            tags: tags,
            type: "link:searchByTags"
        });
    }
}
```

**Programska implementacija 3.3.** LinkActions

### 3.3.3. Stores

Prilikom kreiranja *store* objekta, u ovom slučaju *LinkStore* objekta (Programska implementacija 3.4.), pozvat će se funkcija *register* na *dispatcher* objektu. Toj funkciji se prosljeđuje funkcija čiji je zadatak vršiti provjeru *type* svojstva primljenog objekta (spomenuti objekt se šalje iz *LinkActions* komponente objašnjene u prošlom poglavlju), te u slučaju da primljeni objekt sadrži svojstvo *"link:searchByTags"* bit će pozvana funkcija *searchByTags* funkcija kojoj se prosljeđuju *tag-ovi.* Funkcije *getLinks* i *searchByTags* koriste pomoćni servis pod nazivom *LinkService* (Programska implementacija 3.4.) čija je zadaća obavljanje HTTP poziva prema poslužitelju, te vraćanje podataka dobivenih od poslužitelja.

Kako bi se stvorila veza između *store* i *view* komponente, te ostvarila *Flux* arhitektura, zadužena je funkcija *onChange*, koja odmah poziva *getLinks* funkciju, kako bi se dobili inicijalni podaci za prikaz u *view* komponenti. *OnChange* funkcija također sprema prosljeđenu funkciju pod nazivom *listener*, u *listeners* niz, kako bi *linkStore* komponenta ubuduće mogla obavijestiti *view* komponentu kada dođe do promjene, tj. kada treba obnoviti prikazane podatke u *view* komponenti.

```
var dispatcher = require("../dispatcher");
var linkService = require("../services/linkService")


function LinkStore() {
    var listeners = [];
    function onChange(listener) {
        getLinks(listener);
        listeners.push(listener);
    }
    function getLinks(cb) {
        linkService.getLinks().then(function(res) {
            cb(res);
        });
    }
    function searchByTags(tags) {
      linkService.searchByTags(tags).then(function(res) {
```

```
        listeners.forEach(function(listener) {
            listener(res);
        });
      });
    }

    dispatcher.register(function(payload) {
        var split = payload.type.split(":");
        if (split[0] === "link") {
            switch (split[1]) {
                case "searchByTags":
                    searchByTags(payload.tags);
                    break;
            }
        }
    });
    return {
        onChange: onChange
    }
}
module.exports = LinkStore();
```

**Programska implementacija 3.3.** LinkStore

```
var $ = require("jquery");
var promise = require("es6-promise");
var resourceUrl = "http://localhost:7777/api/links";
module.exports = {
    getLinks: function() {
        var Promise = promise.Promise;
        return new Promise(function(resolve, reject) {
            $.ajax({
                url: resourceUrl + "/front",
                method: "GET",
```

```
                dataType: "json",
                success: resolve,
                error: reject
            });
        });
    },
    searchByTags: function(tags) {
      var Promise = promise.Promise;
      return new Promise(function(resolve, reject) {
          $.ajax({
              url: resourceUrl + "/search",
              data: JSON.stringify(tags),
              contentType: "application/json",
              method: "POST",
              dataType: "json",
              success: resolve,
              error: reject
          });
      });
    }
};
```

**Programska implementacija 3.4.** LinkService

### 3.3.4. Views

*Search* komponenta pri kreiranju poziva *getInitialState* funkciju koja poziva *onChange* funkciju na *linkStore* objektu, te se na taj način stvara poveznica između *store* i *view* komponente. *GetLinksCallback* funkcija će biti pozvana od strane *linkStore* komponente svaki puta kada treba obnoviti prikaz podataka, te će ta funkcija pozvati *this.setState* funkciju.

*SetState* funkcija je posebna *React* funkcija, koja će svaki put kada bude pozvana, pozvati *render* funkciju, koja je zadužena za generiranje prikaza.

Kada korisnik pokrene pretragu, poziva se *search* funkcija, koja poziva *searchByTags* funkciju na *linkStore* objektu, te je na taj način zaokružena *Flux* arhitektura, tj. osiguran je jednosmjeri protok podataka.

```
module.exports = React.createClass({
      getLinksCallback: function(links) {
        this.setState({_links: links});
      },
      getInitialState: function() {
        linksStore.onChange(this.getLinksCallback);
        return {_links: [], searchString: ""}
      },
      searchBarChanged: function(e) {
        e.preventDefault();
        this.setState({
          searchString: e.target.value
        });
      },
      handleEnterPress: function(e) {
        if (e.key === 'Enter') {
          this.search(e);
        }
      },
      search: function(e) {
        e.preventDefault();
        linkActions.searchByTags({tags: _string.words(this.state.searchString)});
      },
      render: function() {
        var list = <div>Enter search tags</div>;
        if (this.state._links != undefined && this.state._links.length > 0) {
            list = <PaginatedLinkList linksPerPage={10} links={this.state._links}/>;
        } else if (this.state._links != undefined && this.state._links.length === 0) {
            list = <div>No links with those tags!</div>
```

```
        }

        return (

            <div className="row">

              <div className="col-md-12 form-group">

                  <label className="control-label" htmlFor="default_tag">Enter search
tags</label>

                  <input type="text" className="form-control" id="search_tags"
name="search_tags" onChange={this.searchBarChanged} onKeyPress={this.handleEnterPress}
placeholder="Enter search tags (no # needed)"/>

                  <br />

                  <button className="btn pull-right" onClick={this.search}>Search</button>

                  <br /><br />

              </div>

              <div className="row">

                {list}

              </div>

            </div>

        );

    }

});
```

**Programska implementacija 3.5.** Search komponenta

# 4. STRUKTURA APLIKACIJE

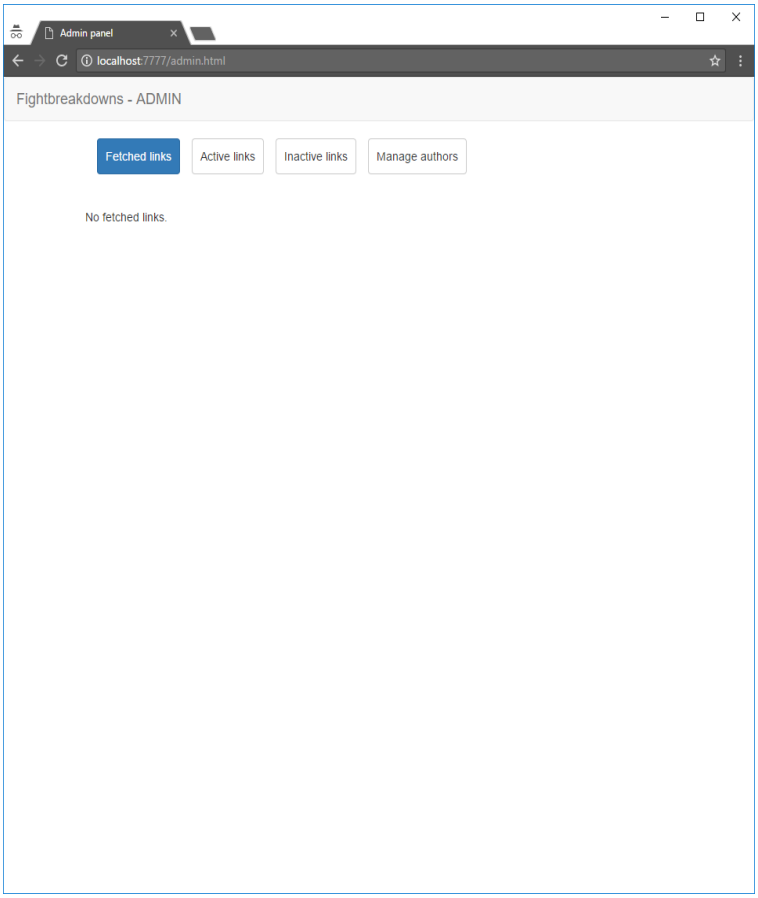
U ovom poglavlju prikazane su sve stranice administratorskog i korisničkog sučelja, te je ukratko opisan način korištenja stranice.

## 4.1. Administratorski dio aplikacije

Administratorsko sučelje aplikacije služi administratoru stranice za praćenje i upravljanje sadržaja prikupljenog s *YouTube* servisa. Postoje četiri podstranice u administratorskom sučelju, od kojega tri služe za upravljanje sadržajem, a jedna za upravljanje autorima, tj. *YouTube* kanalima.

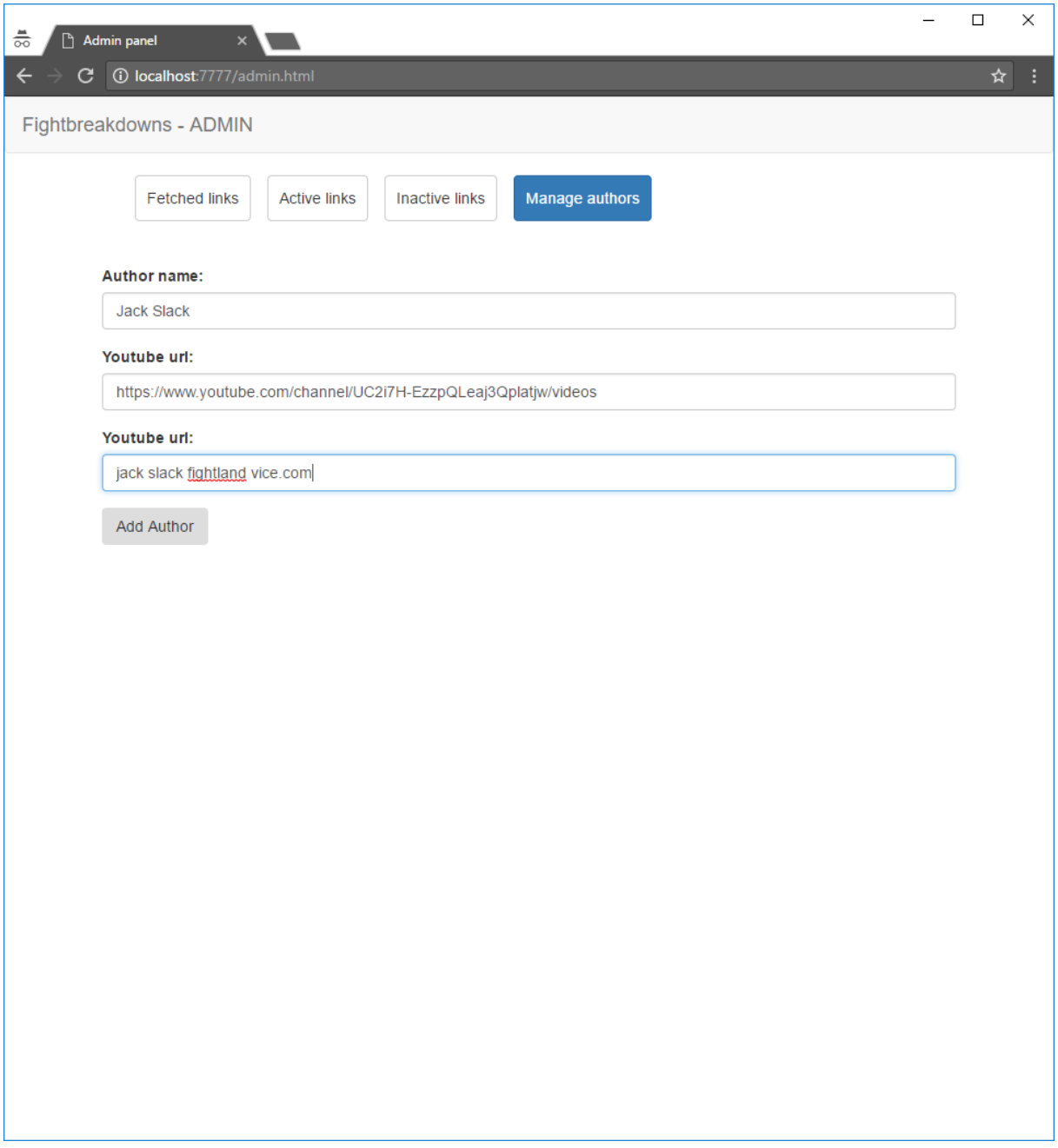
U *Fetched links* dijelu prikazani su novi video isječci koji još nisu raspoređeni u aktivne ili ne aktivne. *Active links* dio prikazuje sve aktivne linkove koji su vidljivi na korisničkoj stranici, dok *Inactive links* dio prikazuje linkove koji nisu aktivni.

Slika 4.1. prikazuje inicijalni izgled stranice kada ne postoje prikupljeni linkovi i autori.

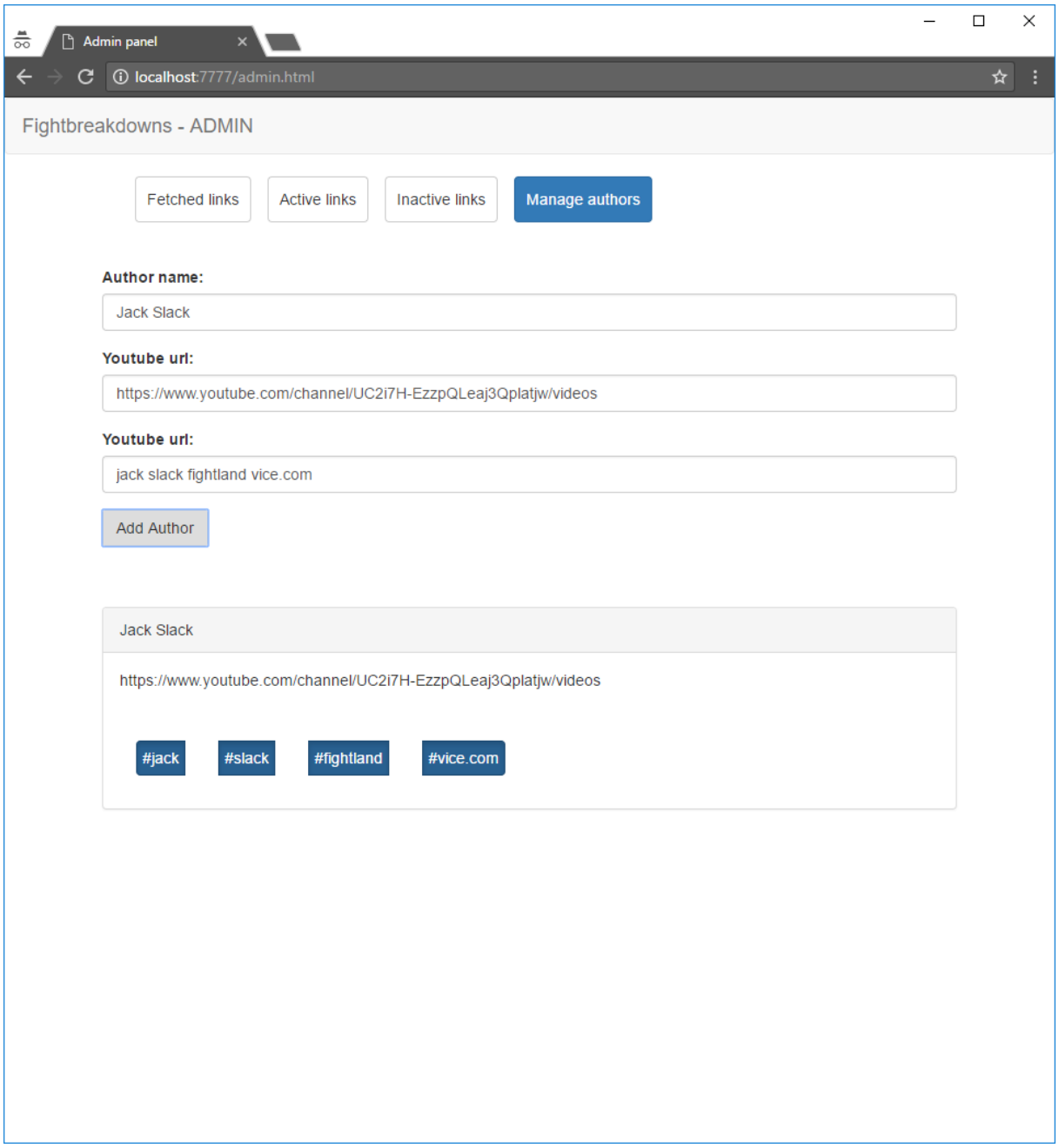



**Sl. 4.1.** Administratorsko sučelje bez sadržaja

Kako bi se započelo automatsko prikupljanje sadržaja, potrebno je unijeti prvog autora. Slika 4.2. prikazuje stranicu za upravljanje autorima. Pri dodavanju autora potrebno je upisati proizvoljno ime koje će označavati autora, poveznicu do autorovog YouTube kanala, te proizvoljne *tagove*, koji će biti automatski odabrani za svaki prikupljeni link toga autora.

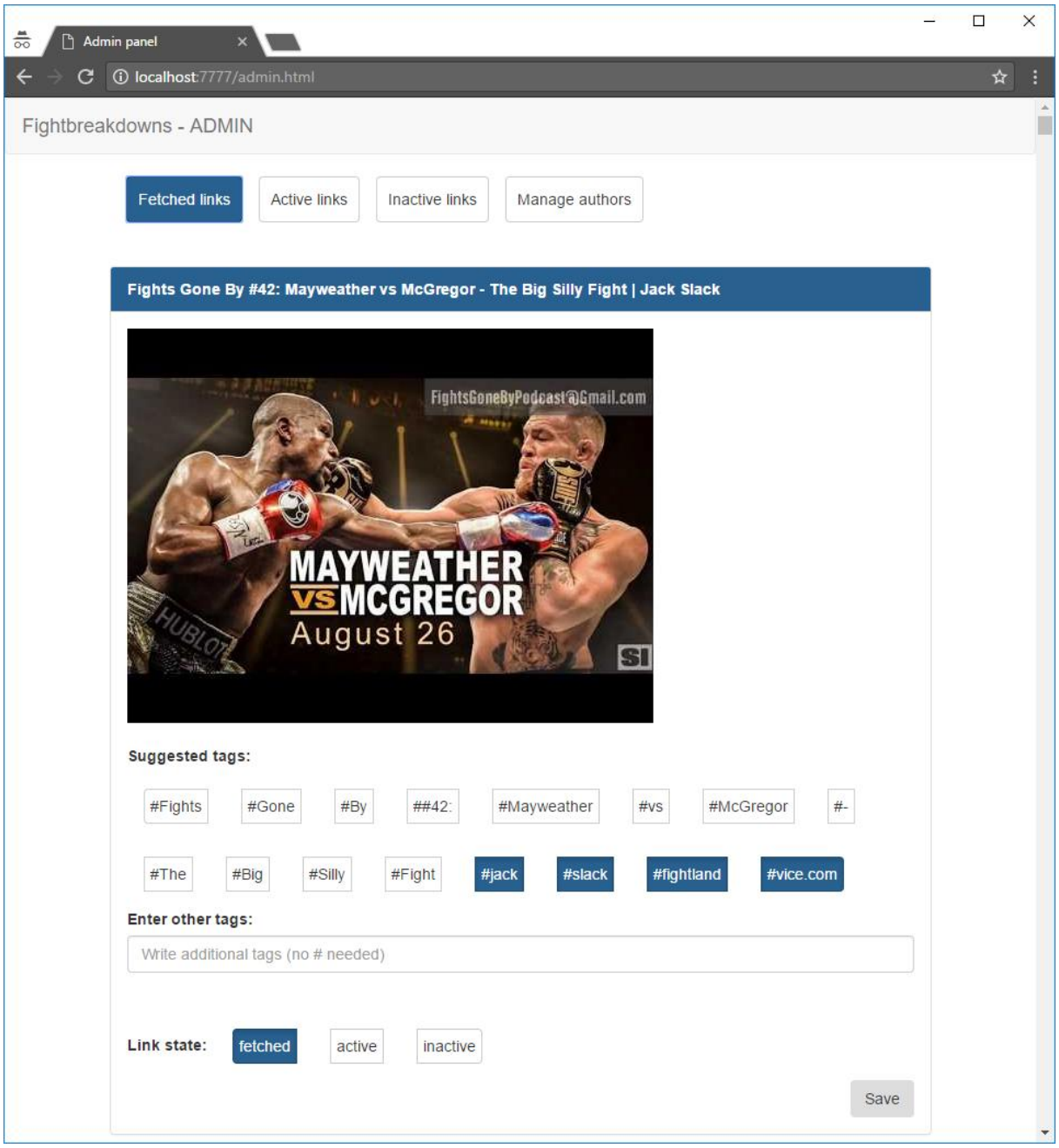



**Sl. 4.2.** Administratorsko sučelje – upravljanje autorima




**Sl. 4.3.** Administratorsko sučelje – dodavanje autora

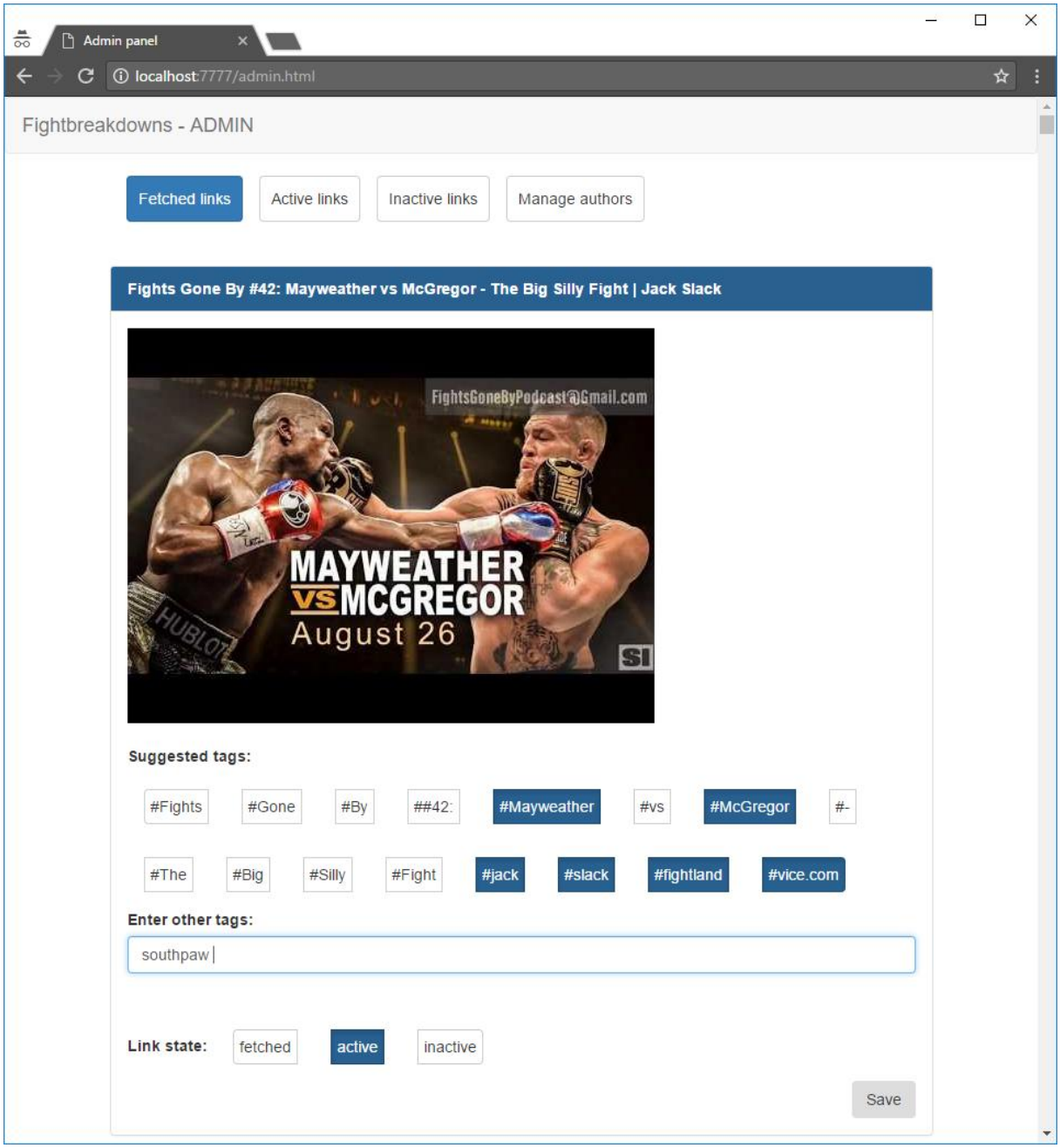
Slika 4.3. prikazuje izgled administratorskog sučelja nakon što je dodan autor. Ispod dijela sučelja gdje se unosi novi autor, prikazan je trenutno jedini uneseni autor. Uz ime autora prikazana je i putanja do *YouTube* kanala i *tagovi*. U slučaju kada bi više autora bilo uneseno, svi autori bili bi prikazani u listi jedan ispod drugoga.



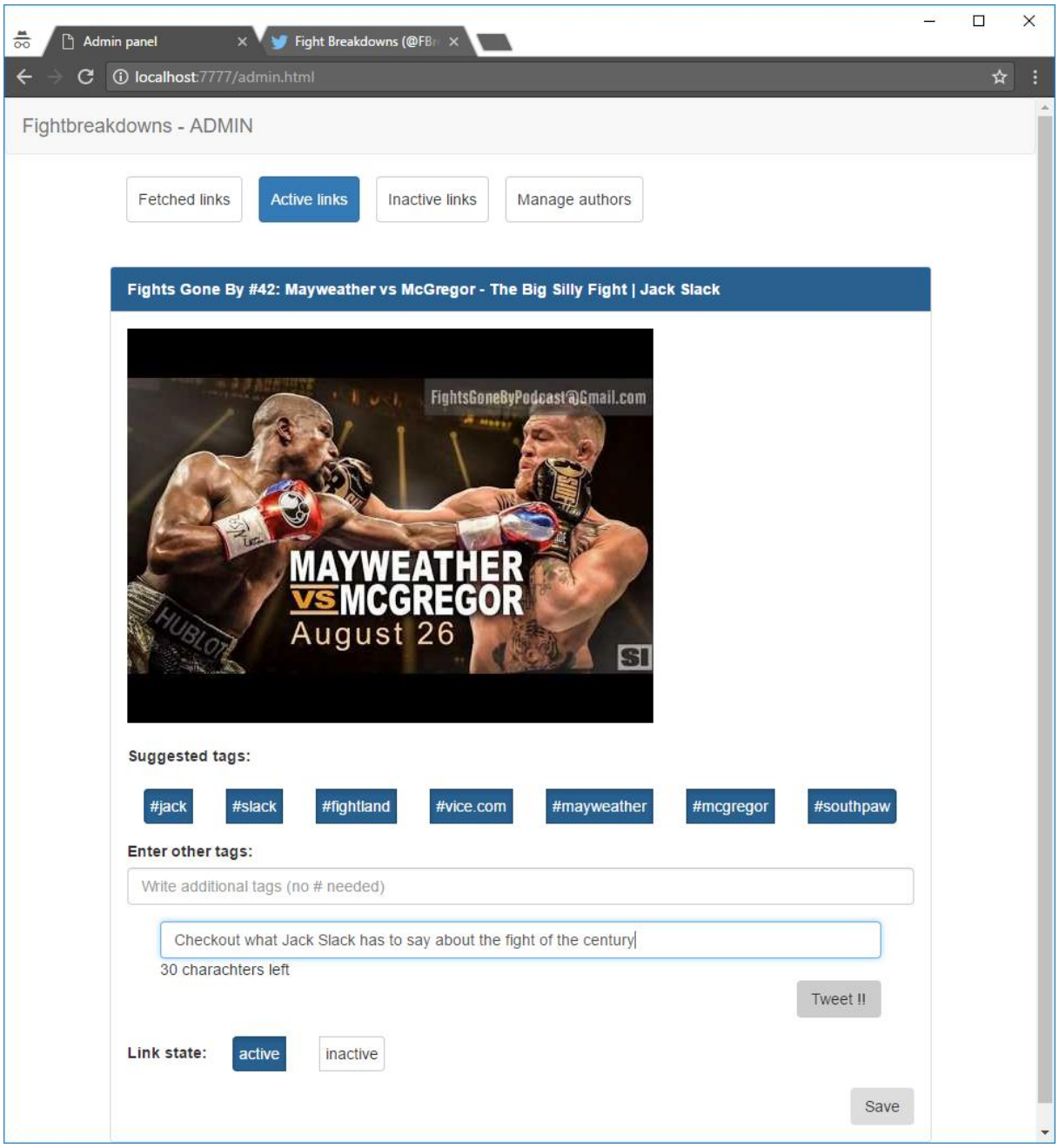

**Sl. 4.4.** Administratorsko sučelje – prikupljeni neobjavljeni sadržaj

Slika 4.4. prikazuje dio administratorskog sučelja s novim prikupljenim i neraspoređenim videima. Administratoru su ponuđeni *tagovi* koji su kreirani pomoću naslova videa, te su automatski odabrani *tagovi* od autora videa, a koje je administrator samostalno upisao kada je dodavao autora. Osim ponuđenih *tagova*, administrator ima mogućnost upisivanja proizvoljnog broja *tagova* za ponuđeni video. Kako bi video bio objavljen u korisničkom sučelju administrator treba odabrati *active* status videa, te pritisnuti gumb *Save*.

Slika 4.5 prikazuje video sadržaj na kojemu je administrator odabrao *tagove*, upisao dodatan *tag* „southpaw“, te odabrao status „active“. Nakon što administrator pritisne gumb „Save“, video će biti vidljiv na „Active links“ podstranici (slika 4.6.).

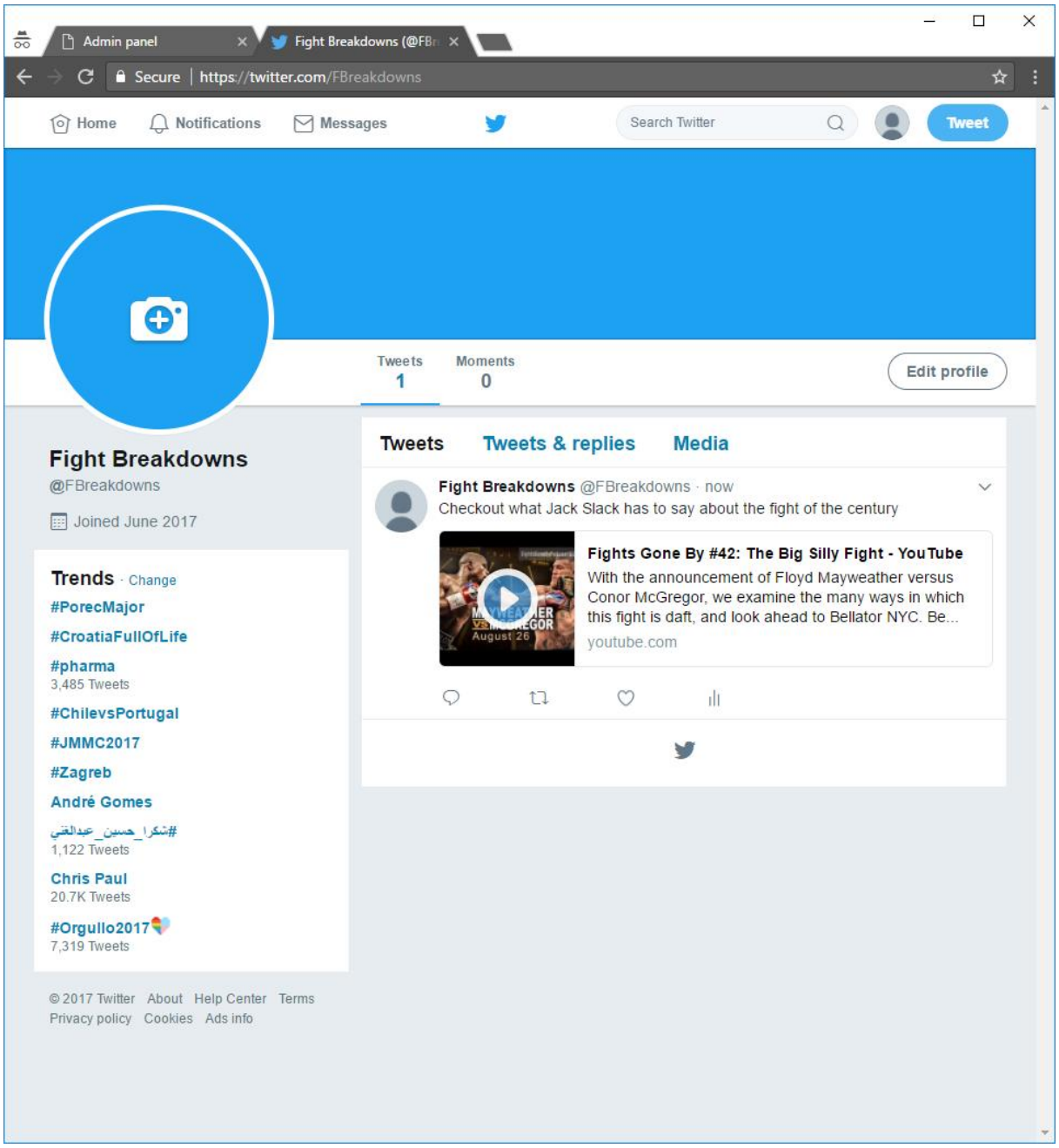



**Sl. 4.5.** Administratorsko sučelje – upravljanje sadržajem




**Sl. 4.6.** Administratorsko sučelje – prikaz aktivnog sadrzaja i postavljanje Tweeta

Slika 4.6. prikazuje dio administratorskog sučelja s aktivnim linkovima. Jednom kada je video aktivan, administrator ga može učiniti neaktivnim, te tada taj video neće biti prikazan u korisničkom sučelju. Kada je video aktivan, administrator ima mogućnost podijeliti taj video na *Twitteru* uz prateću poruku, tako da upiše tekst u za to predviđeni prostor te pritisne gumb „Tweet !!“. Slika 4.7. prikazuje izgled podijeljenog videa i teksta na *Twitteru.*

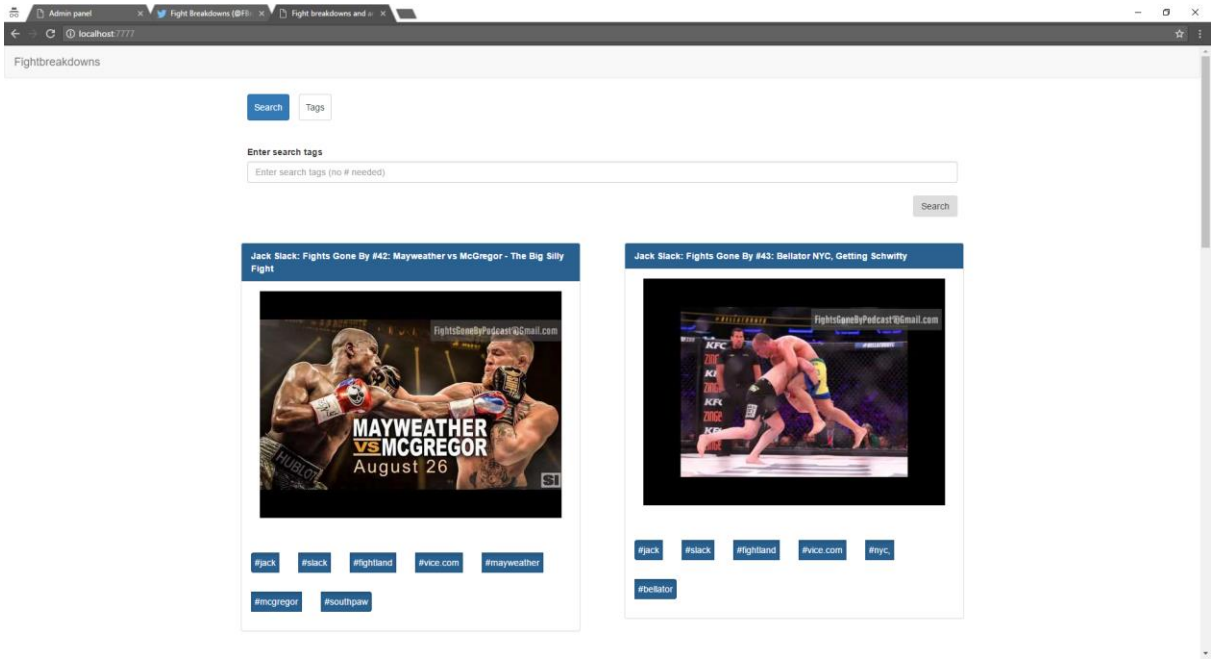



**Sl. 4.7.** Prikaz objavljenog Tweeta

Na *Twitter* stranici vidljiv je tekst koji je unesen u korisničko sučelje na slici 4.6.. Broj unesenih znakova u to polje ograničen je na 140 znakova, isto kao što je prilikom objavljivanja na *Twitter* stranici. Pritiskom na poveznicu na *Twitter* stranici, otvara se video sadržaj na *YouTube* servisu.

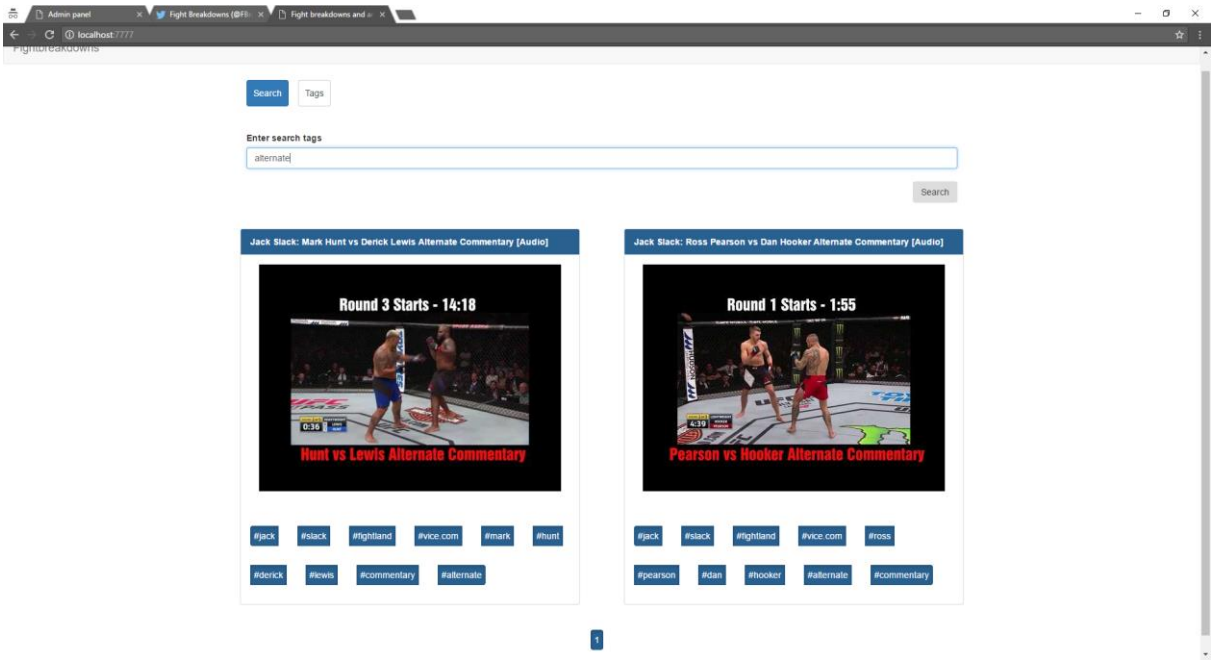
## 4.2. Korisnički dio aplikacije

Korisnički dio aplikacije sastoji se od dvije glavne podstranice, *Search* koja je ujedno i početna stranica, te *Tags* podstranice.



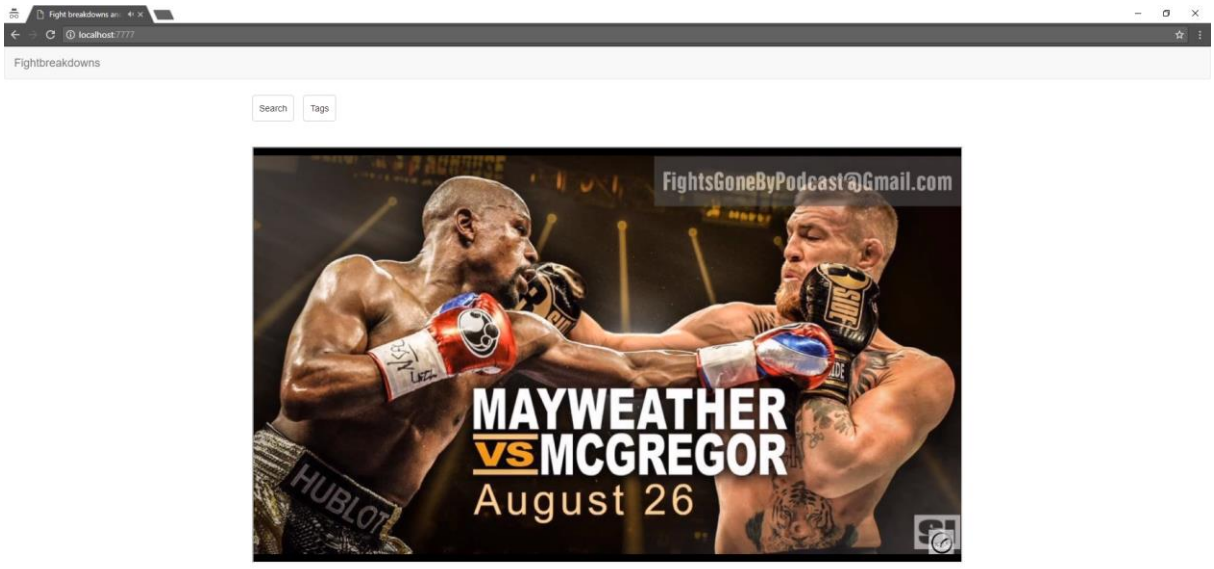

**Sl. 4.8.** Korisničko sučelje – početna stranica

Slika 4.8. prikazuje početnu stranicu koja pri učitavanju sadrži sve dostupne video sadržaje, a prikaz je podijeljen na deset video poveznica po stranici. Korisnik utipkavanjem željenih *tagova* u za to predviđeno polje, te pritiskom na gumb *Search* ili pritiskom *Enter* tipke na tipkovnici filtrira prikazane video sadržaje. Pretraga prema *tagu* „alternate“ prikazana je na slici 4.9..



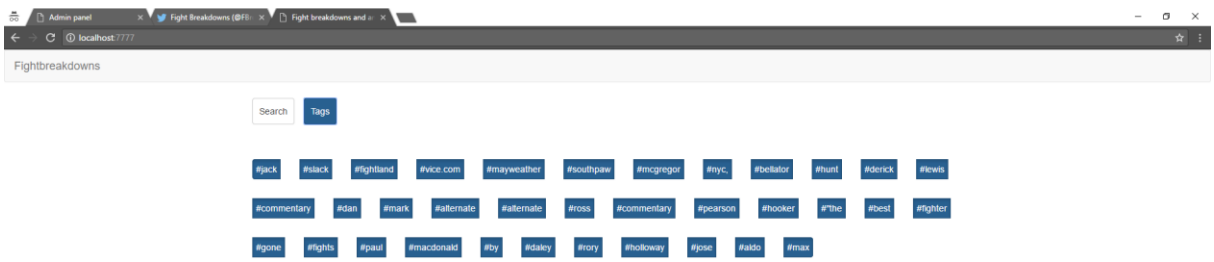

**Sl. 4.9.** Korisničko sučelje – pretraživanje prema tagovima

Pritiskom na video, na svim podstranicama korisničkog sučelja, započinje reprodukcija videa, kao što je prikazano na slici 4.10.

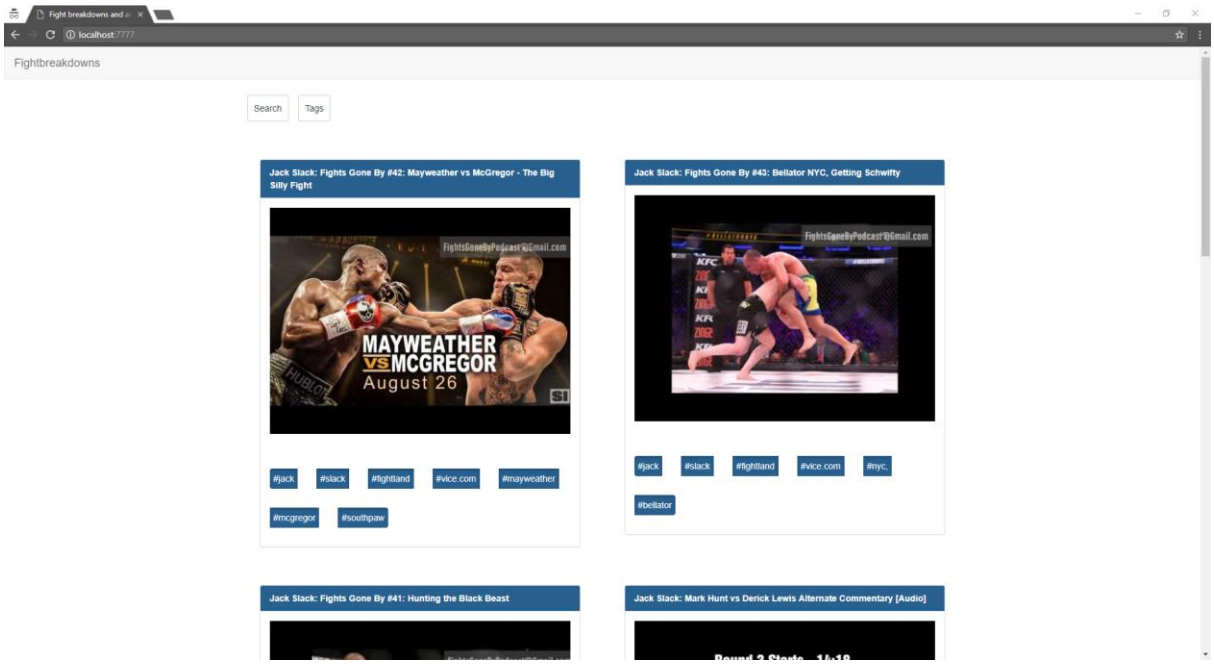



**Sl. 4.10.** Korisničko sučelje – prikaz videa

Podstranica *Tags* (Slika 4.11.), sadrži sve *tagove* koji su barem jednom upotrebljeni na aktivnim videima. Pritiskom na pojedini *tag* prikazuju se svi videi koji sadrže pritisnuti *tag*, slika 4.12. prikazuje primjer rezultata pritiska *tag-a* „vice.com“ .




**Sl. 4.11.** Korisničko sučelje – prikaz svih tagova




**Sl. 4.12.** Korisničko sučelje – rezultati pritiska tag-a „vice.com“

# 5. ZAKLJUČAK

Cilj diplomskog rada bio je kreirati web aplikaciju koja ima sustav automatskog prikupljanja sadržaja s *YouTube* servisa pomoću *JavaScript* skriptnog jezika. Rad se sastoji od tri dijela, u kojemu su objašnjeni poslužiteljski i klijentski dio programskog koda, te korištenje web aplikacije.

Rezultat diplomskog rada je web aplikacija koja je kreirana isključivo s *JavaScript* skriptnim jezikom, a koja se sastoji od administratorskog i korisničkog sučelja. Administratorsko sučelje omogućava upravljanje i objavljivanje prikupljenog sadržaja, dok korisničko sučelje korisnicima omogućava pregled i pretraživanje objavljenog sadržaja.

Kreirana web aplikacija uz preinake programskog koda može postati web aplikacija koja uz prikupljanje ima i automatsko objavljivanje sadržaja, dok bi se ključne riječi automatski generirale iz naslova sadržaja. Također, kreiranjem dodatnih skripti za prikupljanje podataka sadržaj se može prikupljati i s drugih video ili tekstualnih servisa. Takva web aplikacija postaje temelj za kreiranje web stranica čiji se sadržaj bavi određenom tematikom, a čiji bi se sadržaj automatski kreirao.

# LITERATURA


[1] *Part 1: Building web app using react.js, express.js, node.js and mongodb,* https://www.codeproject.com/Articles/1067725/Part-Building-web-app-using-react-js-express-js , 29.6.2017.

[2] *Part 2: Building web app using react.js, express.js, node.js and mongodb,* https://www.codeproject.com/Articles/1068287/Part-Building-web-app-using-react-js-express-js-no , 29.6.2017.

[3] *Getting started with React and Node.js* , https://blog.yld.io/2015/06/10/getting-started-with-react-and-node-js/#.WVV0oIiGOUl , 29.6.2017

[4*] Express.js guide*, https://expressjs.com/en/guide/routing.html, 30.8.2017

[5] *Nodemon*, https://nodemon.io/, 30.8.2017

[6] *What is MongoDB*, https://www.mongodb.com/what-is-mongodb, 30.8.2017

[7] *mongoose* , http://mongoosejs.com/, 30.8.2017

[8] *PhantomJS documentation*, http://phantomjs.org/documentation/, 30.8.2017

[9] *CasperJS documentation*, http://docs.casperjs.org/en/latest/quickstart.html, 30.8.2017

[10] *Cron*, https://www.npmjs.com/package/cron, 30.8.2017

[11] *body-parser*, https://www.npmjs.com/package/body-parser, *30.8.2017*

[12] *HTTP requests*,https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods, 30.8.2017

[13] *ReactJS*, https://facebook.github.io/react/, 30.8.2017

[14] *JSX*, https://jsx.github.io/, 30.8.2017

[15] *Browserify usage*, https://github.com/browserify/browserify#usage, 30.8.2017

[16] *Reactify*, https://github.com/andreypopp/reactify, 30.8.2017

[17] *Gulp documentation,* https://github.com/gulpjs/gulp/blob/master/docs/API.md, 30.8.2017

## SAŽETAK

Tema diplomskog rada je *JavaScript web* aplikacija koja se sastoji od administratorskog i korisničkog sučelja. Poslužitelj *web* aplikacije uz posluživanje sadržaja klijentu ima i zadatak prikupljati sadržaj s *YouTube* servisa. Administratorsko sučelje administratoru omogućava upravljanje prikupljenim sadržajem, dok korisničko sučelje omogućava korisnicima pregled objavljenog sadržaja.

**Ključne riječi:** *JavaScript, YouTube, NodeJs, Express.js, npm*

# ABSTRACT

**Web Application with Automated Gathering of YouTube Content**

The theme of the final paper is a *JavaScript web* application that consists of administrative and user interfaces. Server part of the *web* application has a task to fetch data from *YouTube* along with serving the data to the client. The administrator interface enables administrators to control the data and the public interface enables users to see the gathered and published data.



**Keywords:** *JavaScript, YouTube, NodeJs, Express.js, npm*

# ŽIVOTOPIS

Vjekoslav Getto, rođen 18. 07. 1990. u Osijeku. Živi u mjestu Višnjevac gdje je stekao osnovnoškolsko obrazovanje u Osnovnoj školi Višnjevac. Nakon toga, godine 2005. upisuje Prirodoslovno-matematičku gimnaziju u Osijeku. Godine 2009. upisuje sveučilišni preddiplomski studij Računarstva na Elektrotehničkom fakultetu Sveučilišta Josipa Jurja Strossmayera u Osijeku.

Poseban profesionalni interes iskazuje za Android aplikacije. Govori engleski i njemački jezik. Od prosinca 2015. Vjekoslav Getto zaposlen je kao Android developer u tvrtki CodeConsulting.

## PRILOZI (na CD-u)

- Završni rad u docx formatu
- Završni rad u pdf formatu
- Izvorni kod programa