Swagger code generator za Jcode programski jezik

Ihaz, Mirjam

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni preddiplomski studij Matematika i računarstvo

Mirjam Ihaz

Swagger code generator za JCode programski jezik

Završni rad

Osijek, 2022.

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni preddiplomski studij Matematika

Mirjam Ihaz

Swagger code generator za JCode programski jezik

Završni rad

Mentor:

izv. prof. dr. sc. Domagoj Matijević

Komentor:

Matija Bogdanović, dipl. ing. elektrotehnike

Osijek, 2022.

Swagger code generator za JCode programski jezik

Sažetak

Ovaj rad se bavi razvojem Swagger code generatora za JCode programski jezik. Swagger code generator je open source alat koji iz predloška RESTful API-ija generira izvorni kod JCode aplikacije. Generirani koda se nakon toga može pokrenuti kao aplikacija koja je spremna primati HTTP upite na putanjama definiranim u originalnom predlošku. JCode je programski jezik koji je baziran na Javi, razvijen unutar ENEA-e za jednostavan i modularan razvoj aplikacija u telekomunikacijama. Rad je napravljen u suradnji s tvrtkom ENEA.

Ključne riječi

Swagger, JCode, RESTful API, HTTP protokol

Abstract

This thesis deals with the development of the Swagger code generator for the jcode programming language. The Swagger code generator is an open-source tool that generates JCode application source code from a RESTful API template. The generated code can then be run as an application ready to receive HTTP requests on the paths defined in the original proposal. JCode is a programming language based on Java, developed within ENEA for simple and modular development of applications in telecommunications. This graduation thesis was made in cooperation with ENEA.

Key words

Swagger, JCode, RESTful API, HTTP protocol

Sadržaj

1 | Uvod

Swagger je open source skup pravila, specifikacija i alata za razvoj i opisivanje RESTful API-ija [1]. Swagger pomaže korisnicima da izgrade, dokumentiraju, testiraju i koriste RESTful web usluge. Najvažnija značajka OpenAPI-ija je kada je napisana specifikacija Swagger alati mogu potaknuti razvoj API-ija u više smjerova. Korisnici koriste Swagger Codegen za generiranje poslužitelja za vlastiti API, a jedino što preostaje je implementirati logiku poslužitelja. Swagger Codegen možemo koristiti za generiranje klijentskih biblioteka za API na više od 40 jezika, a u ovom projektu smo dodali još jedan jezik, a to je JCode. Dakle, programerima omogućuju uštedu vremena i energije za razvoj RESTful API-ija. Može se koristiti s pristupom razvoja API-ja odozgo prema dolje i odozdo prema gore. U top-down, ili design-first metodi, Swagger se koristiti za dizajn API-ja prije nego što se napiše kod. U metodi odozdo prema gore Swagger uzima kod napisan za API i generira dokumentaciju. Projekt Swagger API kreirao je 2011. Tony Tam tijekom razvoja alata za web stranicu rječnika Wordnik. Osmišljen je kako bi se olakšala automatizacija API-ja i njegove dokumentacije. Projekt je zatim postao open source gdje je stekao popularnost među tvrtkama i programerima. Godine 2015. tvrtka koja je održavala Swagger, SmartBear Software, pomogla je u osnivanju OpenAPI inicijative, organizacije koju sponzorira Linux Foundation. Godinu dana kasnije, Swagger je preimenovan u OpenAPI Specification i premješten je u novi GitHub repozitorij. Swagger je trenutno najveći okvir za dizajniranje API-ja sa zajedničkim jezikom [2]. U sljedećim poglavljima ćemo reći što je to RESTful API, HTTP protokol, te ćemo na temelju projekta obijasniti kako funkcionira Swagger.

2 | Tehnički aspekti Swagger koncepta

2.1 Što je RESTful API?

REST API, poznat kao RESTful API, je sučelje za programiranje aplikacije, takvo da zadovoljava REST arhitekturu i omogućuje interakciju s RESTful web uslugama. REST (engl. *Representational state transfer*) je kratica za reprezentativni prijenos stanja, a kreirao ga je znanstvenik Roy Fielding. API (engl. *application programming interface*) je skup definicija i protokola za izgradnju i integraciju aplikacijskog softvera. API se može zamisliti kao posrednika između korisnika ili klijenata i resursa ili web usluga koje žele dobiti. Prednost API-ja je što ne moramo znati specifičnost predmemoriranja, kako se resurs dohvaća ili odakle dolazi. REST nije protokol ili standard, već je skup ograničenja arhitekture. Kada klijent zatraži zahtjev putem RESTful API-ja, on prenosi prikaz stanja resursa podnositelju zahtjeva ili krajnjoj točki. Informacije se isporučuju putem HTTP-a kao JSON, HTML, XLT ili čisti tekst. JSON je najpopularniji format datoteke. API smatramo RESTful ako zadovoljava sljedeće kriterije:

- Arhitektura klijent-poslužitelj mora biti sastavljena od klijenata, poslužitelja i resursa, sa zahtjevima kojima se upravlja putem HTTP-a.
- Informacije o klijentu se ne pohranjuju između zahtjeva za dohvaćanje i svaki zahtjev je odvojen i nepovezan.
- Podaci koji se mogu predmemorirati pojednostavljuju interakcije klijent-poslu -žitelj.
- Uniformno sučelje između komponenti je tako da se informacije prenose u standardnom obliku. To zahtijeva sljedeće:
  - zatraženi resursi su prepoznatljivi i odvojeni od prikaza poslanih klijentu.
  - resursima klijent može manipulirati putem reprezentacije koju prima jer reprezentacija sadrži dovoljno informacija za to.
  - samoopisne poruke vraćene klijentu imaju dovoljno informacija za opis kako bi ih klijent trebao obraditi.

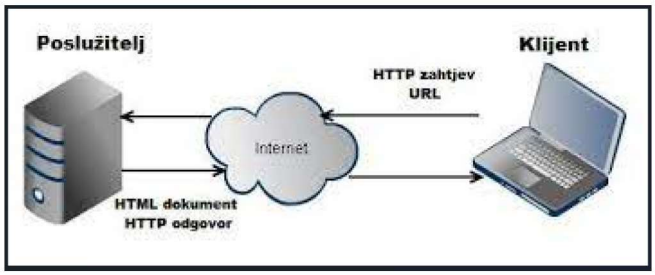
  - hipertekst/hipermedij je dostupan, što znači da bi nakon pristupa resursu klijent trebao moći koristiti hiperveze za pronalaženje svih drugih trenutno dostupnih radnji koje može poduzeti.

- Slojeviti sustav koji organizira svaku vrstu poslužitelja, uključuje dohvaćanje traženih informacija u hijerarhije, koje su nevidljive klijentu.
- Kod na zahtjev, što je mogućnost slanja izvršnog koda s poslužitelja klijentu kada se to zatraži, proširujući funkcionalnost klijenta.[3]

2.2 HTTP protokol

HTTP, engl. Hyper Text Transfer Protocol, je glavna i najčešća metoda prijenosa informacija na Webu [4]. Osnovna namjena ovog protokola je što omogućuje objavljivanje i prezentacije HTML dokumenata, tj. web stranica. HTTP protokol definira kako web klijent i poslužitelj međusobno komuniciraju. Detaljno specificira zahtjeve koje klijent smije zatražiti od poslužitelja i kako poslužitelj treba odgovoriti. HTTP klijent, kao što je web preglednik najčešće inicira prijenos podataka nakon što uspostavi TCP vezu s udaljenim web serverom na određenom portu. HTTP koristi specifične metode zahtjeva kako bi izvršio različite zadatke. Svi HTTP poslužitelji koriste metode GET i HEAD, ali ne podržavaju svi ostale metode zahtjeva:

- GET traži podatak iz određenog resursa.
- HEAD zahtjeva određeni resurs bez sadržaja tijela.
- POST dodaje sadržaj, poruke ili podatke na novu stranicu unutar postojećeg web izvora.
- PUT izravno mijenja postojeći web resurs ili stvara novi URI ako je potrebno.
- PATCH djelomično mijenja web resurs.
- DELETE briše navedeni resurs.[5]



Slika 2.1: Slika prikazuje protokol [6]

2.3 Što je SWAGGER?

Swagger je skup open-source alata izgrađenih oko specifikacije OpenAPI. Glavni Swagger alati uključuju:

- Swagger Editor – uređivač OpenAPI definicija izravno iz internet preglednika.
- Swagger UI – prikazuje OpenAPI definicije kao interaktivnu dokumentaciju.
- Swagger Codegen – generira predloške poslužitelja i klijentske biblioteke iz OpenAPI definicije
- Swagger Editor Next (beta) – uređivač temeljen na internet pregledniku u kojem je moguće pisati i pregledavati OpenAPI i AsyncAPI definicije.
- Swagger Core – biblioteke povezane s Javom za stvaranje, korištenje i rad s OpenAPI definicijama.
- Swagger Parser – samostalna biblioteka za raščlanjivanje OpenAPI definicija.
- Swagger APIDom – pruža jednu jedinstvenu strukturu za opisivanje API-ja u različitim jezicima opisa i formatima serijalizacije.[7]

2.4 OpenAPI

Specifikacija OpenAPI, odnosno Swagger specifikacija je opis API-ja za REST API-je. OpenAPI datoteka omogućuje da opišete svoj cijeli API, uključujući:

- Dostupne putanje i operacije na svakoj putanji (GET /user, POST /user).
- Parametri operacije Ulaz i izlaz za svaku operaciju.
- Metode provjere autentičnosti.
- Kontakt informacije, licenca, uvjeti korištenja i druge informacije.
- API specifikacije mogu se napisati u YAML ili JSON datoteci.[8]

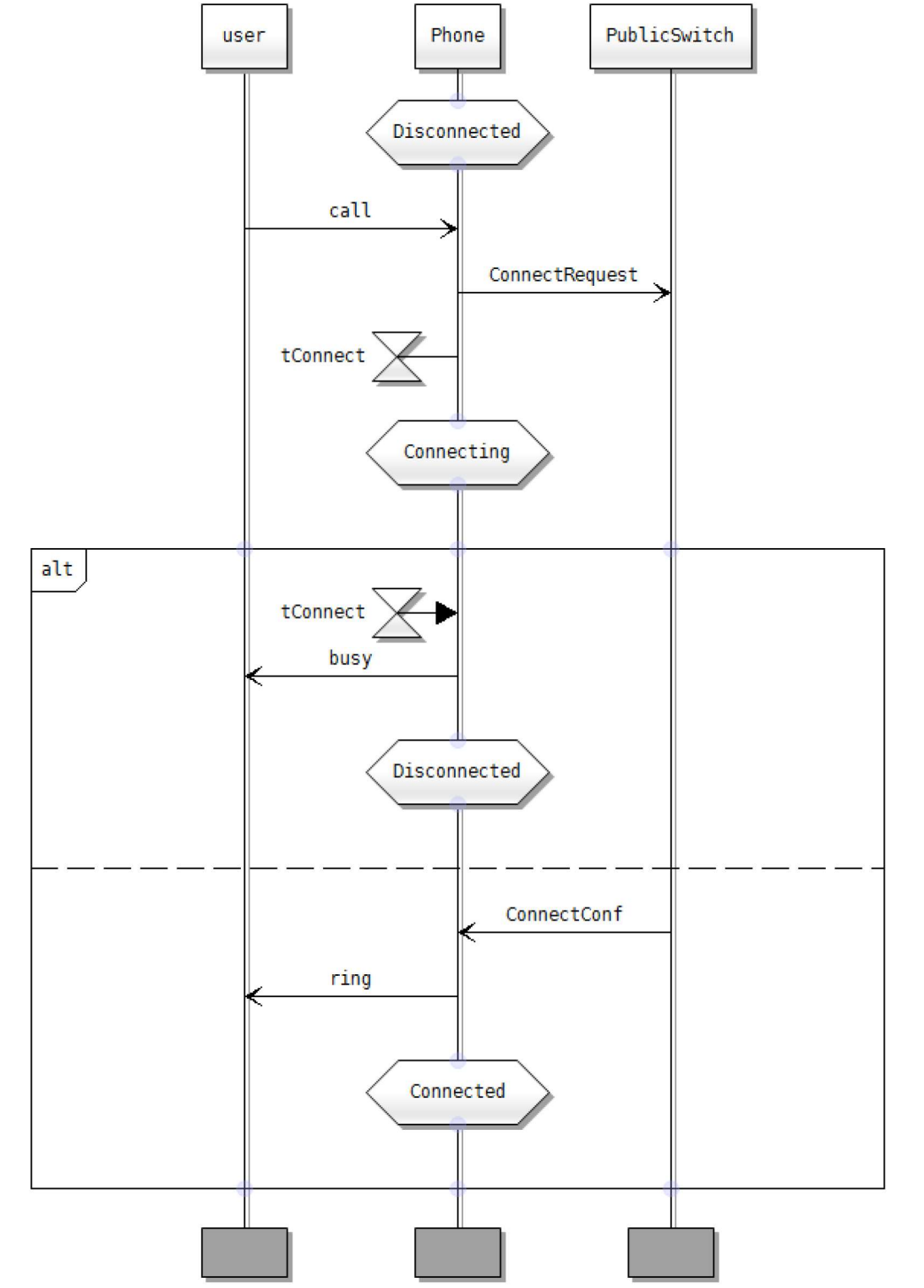
2.5 JDK

JDK (engl. *Java Development Kit*) jedan je od tri temeljna paketa koji se koriste u Java programiranju, zajedno s JVM (Java Virtual Machine) i JRE (Java Runtime Environment)[9]. Java Virtual Machine je virtualni stroj koji računalu omogućuje pokretanje Java programa, kao i programa napisanih na drugim jezicima koji su također kompajlirani u Java bajt kod [10]. Java Runtime Environment ili JRE je softverski sloj koji radi povrh softvera operacijskog sustava računala i pruža biblioteke klasa i druge resurse koje određeni Java program treba pokrenut [11]. JDK je okruženje za razvoj softvera koje se koristi za razvoj Java aplikacija. JDK

ima zbirku alata za programiranje koja uključuje: appletviewer, apt, extcheck, idlj, jabswitch, java, javac, javadoc, jar, javafxpackager, jarsigner, javah, javap, javaws, jdb, jinfo, jmap, jmc, jpackage, jps, jshell itd. Ovdje trebamo istaknuti .jar datoteku koja će nam biti potrebna prilikom generiranja koda. JAR datoteka omogućuje Java runtimerima da učinkovito implementiraju cijelu aplikaciju, uključujući njene klase i njima pridružene resurse, u jednom zahtjevu[12].

2.6 MSC

Grafikon sekvence poruka, MSC (engl. *Message sequence chart*), je dijagram interakcije iz obitelji SDL standardiziran od strane Međunarodne telekomunikacijske unije. MSC jezik je jednostavan za naučiti, koristiti i interpretirati. S drugim jezicima može se koristiti za podršku metodologijama za specifikaciju sustava, dizajn, simulaciju, testiranje i dokumentaciju. MSC, odnosno diagram slijeda poruka, osigurava jezik za praćenje za specifikaciju i opis komunikacijskog ponašanja komponenti sustava i njihovog okruženja putem razmjene poruka. Na sljedećoj slici možemo vidjeti jedan primjer dijagrama. Dijagram prikazuje tri entiteta. Pri pokretanju telefon je isključen. Korisnik pokušava uspostaviti vezu. Zahtjev za povezivanje šalje se preklopniku i pokreće se tajmer. Imamo dva moguća odgovora, prvi je da se tajmer gasi jer prekidač nije odgovorio i telefon se vraća u isključeno stanje, a drugi da prekidač odobrava vezu i poziv je uspostavljen.[13]



Slika 2.2: Slika prikazuje dijagram [14]

3 | Praktični zadatak

Cilj ovog projekta bio je razviti Swagger code generator za JCode programski jezik. JCode je programski jezik koji je baziran na Java-i, razvijen unutar ENEA-e. U ovom poglavlju ćemo opisati tijek razvoja projekta.

3.1 Osnovna struktura OpenAPI-ija

Specifikacija OpenAPI je "jezično-agnostičko sučelje za RESTful API-je" koje omogućuje programerima da raspravljaju o REST API-jima tako da svi razumiju. Specifikacija omogućuje razvojnim programerima da stvore ugovor koji definira kako će API raditi i što treba raditi prije nego što se napiše kod. U ovom projektu je OpenAPI specifikacija napisana u YAML datoteci. Kako bi razumjeli OpenAPI specifikaciju po uzoru na "Petstore" aplikaciju, koja se nalazi u Swagger Editoru, smo izgenerirali vlastiti kod za "todo" aplikaciju. Izgenerirana je kao python-flask server te pokrenuta u PyCharmu. Pogledat ćemo na primjeru "todo" aplikacije osnovnu strukturu.

- Svaka API definicija mora sadržavati verziju OpenAPI Specifikacije na kojoj se ova definicija temelji. Verzija OpenAPI definira cjelokupnu strukturu API definicije, što se može dokumentirati i kako to dokumentirati. Ovdje vidimo kao primjer "2.0" verziju. Odjeljak s informacijama sadrži informacije o API-ju: naslov, opis, koji nije obavezan te verziju. Title je ime aplikacije. Description je proširena informacija o API-ju. Verzija je proizvoljni niz koji navodi verziju API-ja. Može se koristiti semantičku verziju kao što je major.minor.patch ili proizvoljni format poput 1.0-beta ili 2016.11.15. Info također podržava druge ključne riječi za podatke o kontaktu, licencu, uvjete usluge i druge detalje.

```yaml
swagger: "2.0"
info:
  description: "Todo"
  version: "1.0.0"
  title: "Swagger Todo"
```

Slika 3.1: Slika prikazuje info objekt i API definiciju

- Osnovni URL za sve API pozive definiran je pomoću shema, hosta i basePath-a. Sve API putanje relativne su u odnosu na osnovni URL.

```yaml
host: "todo.swagger.io"
basePath: "/v2"
tags:
- name: "todo"
  description: "Everything about todo"
  externalDocs:
    description: "Find out more"
    url: "http://swagger.io"
schemes:
- "https"
- "http"
```

Slika 3.2: Slika prikazuje osnovni URL

- Odjeljak paths definira pojedinačne krajnje točke (putanje) u API-ju i HTTP metode (operacije) koje te krajnje točke podržavaju.

```yaml
paths:
  /todo:
    post:
      tags:
      - "todo"
      summary: "Create todo"
      description: "This can only be done by the logged in todo."
      operationId: "create_todo"
      produces:
      - "application/xml"
      - "application/json"
      parameters:
      - in: "body"
        name: "body"
        description: "Created todo object"
        required: true
        schema:
```

Slika 3.3: Slika prikazuje odjeljak paths

- Operacije mogu imati parametre koji se mogu proslijediti putem URL putanje (/todo/task), niza upita, zaglavlja i tijela zahtjeva. Mogu se definirati vrste parametara, format, jesu li obavezni ili izborni i druge detalje:

```yaml
/todo/{task}:
  get:
    tags:
    - "todo"
    summary: "Get task"
    description: ""
    operationId: "get_task"
    produces:
    - "application/xml"
    - "application/json"
    parameters:
    - name: "task"
      in: "path"
      description: "The task that needs to be fetched. "
      required: true
      type: "string"
    responses:
```

Slika 3.4: Slika prikazuje primjer parametara koji su proslijeđeni putem URL putanje

- Za svaku operaciju može se definirati moguće statusne kodove, poput 200 OK ili 404 Not Found, i shemu tijela odgovora. Sheme se mogu definirati inline ili se mogu referencirati iz vanjske definicije preko ref. [15]

```yaml
get:
  tags:
  - "todo"
  summary: "Get task"
  description: ""
  operationId: "get_task"
  produces:
  - "application/xml"
  - "application/json"
  parameters:
  - name: "task"
    in: "path"
    description: "The task that needs to be fetched. "
    required: true
    type: "string"
  responses:
    "200":
      description: "successful operation"
      schema:
        $ref: "#/definitions/Todo"
    "400":
      description: "Invalid task supplied"
    "404":
      description: "Task not found"
  x-swagger-router-controller: "swagger_server.controllers.todo_controller"
```

Slika 3.5: Slika prikazuje tijelo odgovora

3.2 Swagger Codegen

Swagger Codegen je generator otvorenog izvornog koda za izradu poslužiteljskih dopuna i klijentskih SDK-ova izravno iz Swagger definiranog RESTful API-ja. Izvorni kod za Swagger Codegen može se pronaći na GitHubu. U ovom projektu smo prvo klonirali projekt sa GitHuba lokalno, a nakon toga pokrenuli smo ga naredbom

- "mvn clean package".

Sljedećom naredbom dobijemo predložak koji će outputati u MyLibrary, dobijemo jar datoteku. Ova .jar datoteka je alat naredbenog retka koji pruža sučelje koje je potrebno za korištenje Swagger Codegena kada budemo generiranli kod.

- "java -jar modules/swagger-codegen-cli/target/swagger-codegen-cli.jar meta -o output/myLibrary -n myClientCodegen -p com.my.company.codegen ".

Zatim dobijemo projekt MyLibrary koji također pokrečemo Mavenom:

- "mvn clean package".

U myLibrary dobijemo mustache datoteku. Trebamo napisati api.mustache file kao bi izgenerirali myrest.yaml, odnosno specifikaciju za kod JCode aplikacije. Na sljedećim slikama možemo vidjeti primjer myrest.yaml koda, odnosno koda naše aplikcije.

```
openapi: '3.0.0'
info:
  version: '1.0'
  title: Todo list
  description:  Definition file for a Todo list
  termsOfService: 'http://api.domain.com/terms/'
servers:
  - url: '{scheme}://{address}:{port}/{version}/todos'
    variables:
        version:
          description: '..'
          enum:
            - 'v0.1'
            - 'v1'
          default: 'v1'
        port:
          description: '..'
          enum:
            - '19003'
          default: '19003'
        scheme:
          description: '..'
          enum:
            - 'http'
            - 'https'
          default: 'https'
        address:
          description: '..'
          enum:
            - '192.168.0.2'
            - 'todo.example.com'
            - '127.0.0.1'
          default: '127.0.0.1'
```

Slika 3.6: Primjer myrest.yaml koda

```
paths:
  '/':
    post:
      summary: Create Todo task
      x-signal: CreateTodoReq
      operationId: CreateTodoReq
      tags:
        - Todo task
      requestBody:
        content:
          application/json:
            schema:
              type: object
              required:
                - title
              properties:
                title:
                  type: string
                description:
                  type: string
        required: true
      responses:
        '201':
          description: Successfuly created task
          content:
            application/json:
              schema:
                type: object
                properties:
                  title:
```

Slika 3.7: Primjer putanje myrest.yaml koda

Nakon što je mustache napisan, koji se nalazi u projektu MyLibrary, ponovno ga pokrećemo Mavenom. Zatim u Swagger Codgenu sljedećom dnaredbom dobijemo projekt MyClient u kojem se nalazi MyserverApi.sample u kojem dobijemo naš izgenerirani kod.

- "java -cp 'modules/swagger-codegen-cli/target/swagger-codegen-cli.jar out -put/myLibrary/target/myClientCodegen-swagger-codegen-1.0.0.jar' io.sw -agger.codegen.SwaggerCodegen generate -l myClientCodegen -i https:// petstore.swagger.io/v2/swagger.json -o myClient"

3.3 Mustache

Brkovi su sintaksa „šablona bez logike“ jer nema if ili else nardbi te for petlji. Umjesto toga postoje samo oznake. Može se koristiti za HTML, konfiguracijske datoteke, izvorni kod, bilo što. Mustache je sustav web predložaka s implementacijama dostupnim za ActionScript, C++, Clojure, CoffeeScript, ColdFusion, Common Lisp, Crystal, D, Dart, Delphi, Elixir, Erlang, Fantom, Go, Haskell, Io, Java,

JavaScript, Julia, Lua , .NET, Objective-C, OCaml, Perl, PHP, Pharo, Python, R, Racket, Raku, Ruby, Rust, Scala, Smalltalk, Swift, Tcl, CFEngine i XQuery. Radi proširivanjem oznaka u predlošku pomoću vrijednosti navedenih u hash-u ili objektu [16]. Mustache funkcionira tako da pomoću oznaka u predlošku napišemo specifičnu sintaksu kojom ćemo izgenerirati traženi kod. Na sljedećoj slici 8.8 možemo vidjeti primjer dijela objekta za JCode aplikaciju koju ćemo izgenerirati.

```
"operation" : [ {
  "responseHeaders" : [ ],
  "hasProduces" : true,
  "hasParams" : true,
  "hasMore" : true,
  "isResponseBinary" : false,
  "path" : "/task",
  "operationId" : "CreateTask",
  "httpMethod" : "POST",
  "summary" : "Create Task",
```

Slika 3.8: Primjer dijela objekta

Na slikama 8.9.ii 8.10. možemo vidjeti kako izgleda mustache kod. Dio koda koji započinje s "operations" prolazi kroz sve operacije i za svaku operaciju uzima njezin "operatioId", odnosono Id. Dakle, na tom principu funkcionira pisanje mustache koda.

```
Application {
    TestApp {
        Applications {
            TestApp {
                Languages { java }
                Rules {
                {{#operations}}
                    {{#operation}}
                    {{vendorExtensions.x-signal}}Req = {
                        h_{{vendorExtensions.x-signal}}()
                    }.
                    {{/operation}}
                {{/operations}}
                }
                Handler {
                {{#operations}}
                    {{#operation}}
                    h_{{vendorExtensions.x-signal}} {
                        JCode = {
                            /myRes = getRes(/Request);
                            {{#responses}}{{#@first}}/myRes/UriParams/HttpStatusCode
                                 = {{code}};{{/@first}}{{/responses}}
                            /myRes/Body/handler = "h_{{vendorExtensions.x-signal}}";
                            reply /myRes;
                        }.
                    }
                    {{/operation}}
                {{/operations}}
                }
```

Slika 3.9: api.mustache

```
                Tests {
                {{#operations}}
                    {{#operation}}
                    test-{{vendorExtensions.x-signal}} {
                        Kind = Test
                        MSC = {
                            ##User this
                            User->this:{{vendorExtensions.x-signal}}Req:
                                Headers={Method='{{httpMethod}}',Scheme='https',
                                Path='{{testPath}}',Content-Type='application/json',
                                Authority='10.14.45.214:19000',User-Agent='curl/7.60.0',
                                Accept='*/*',Host='10.14.45.214:19000'},
                                UriParams={Method='{{httpMethod}}',Uri='{{testPath}}',
                                Question='',Protocol='HTTP',Version='2'},
                                QueryParams={},PathParams={},Body={}
                            User<-this:{{vendorExtensions.x-signal}}Res:
                                UriParams={HttpStatusCode=
                                {{#responses}}{{#@first}}{{code}}{{/@first}}{{/responses~}} }
                        }.
                    }
                    {{/operation}}
                {{/operations}}
                }
                FormatVersion = 2
            }
        }
    }
}
```

Slika 3.10: api.mustache

3.4 Testiranje

Generianjem mustache koda smo dobili MyserverApi.sample datoteku koju možemo vidjeti na sljedećim slikama, odnosno naš izgenerirani kod. Zatim smo taj kod pomoću Dockera testirali. Docker omogućuje jednostavno sučelje koje omogućuje upravljanje kontenjerima, aplikacijama i slikama izravno s vlastitog računala bez potrebe za korištenjem CLI-ja za izvođenje osnovnih radnji. Tehnologija Docker koristi jezgru Linuxa za odvajanje procesa kako bi se mogli izvoditi neovisno. Docker omogućuje skidanje dijela aplikacije radi ažuriranja ili popravka, bez potrebe za skidanjem cijele aplikacije. Uz ovaj pristup mogu se dijeliti procesi između više aplikacija [17]. U progamu Visual Studio Code smo otvorili relgui aplikaciju koja sardži TestApp i RESTProfile datoteke. Aplikaciju smo zapravo stavili u linux kontenjer te testirali prima li http zahtjeve.

```
Application {
    TestApp {
        Applications {
            TestApp {
                Languages { java }
                Rules {
                    CreateTodoReqReq = {
                        h_CreateTodoReq()
                    }.
                    DeleteTodoReqReq = {
                        h_DeleteTodoReq()
                    }.
                    PatchTodoReqReq = {
                        h_PatchTodoReq()
                    }.
                    ReadTodoReqReq = {
                        h_ReadTodoReq()
                    }.
                    ReadTodosReqReq = {
                        h_ReadTodosReq()
                    }.
                }
                Handler {
                    h_CreateTodoReq {
                        JCode = {
                            /myRes = getRes(/Request);
                            /myRes/UriParams/HttpStatusCode = 201;
                            /myRes/Body/handler = "h_CreateTodoReq";
                            reply /myRes;
                        }.
                    }
                    h_DeleteTodoReq {
                        JCode = {
                            /myRes = getRes(/Request);
                            /myRes/UriParams/HttpStatusCode = 200;
                            /myRes/Body/handler = "h_DeleteTodoReq";
                            reply /myRes;
                        }.
                    }
                    h_PatchTodoReq {
                        JCode = {
                            /myRes = getRes(/Request);
                            /myRes/UriParams/HttpStatusCode = 200;
                            /myRes/Body/handler = "h_PatchTodoReq";
                            reply /myRes;
                        }.
                    }
                    h_ReadTodoReq {
                        JCode = {
                            /myRes = getRes(/Request);
                            /myRes/UriParams/HttpStatusCode = 200;
```

Slika 3.11: MyserverApi.sample

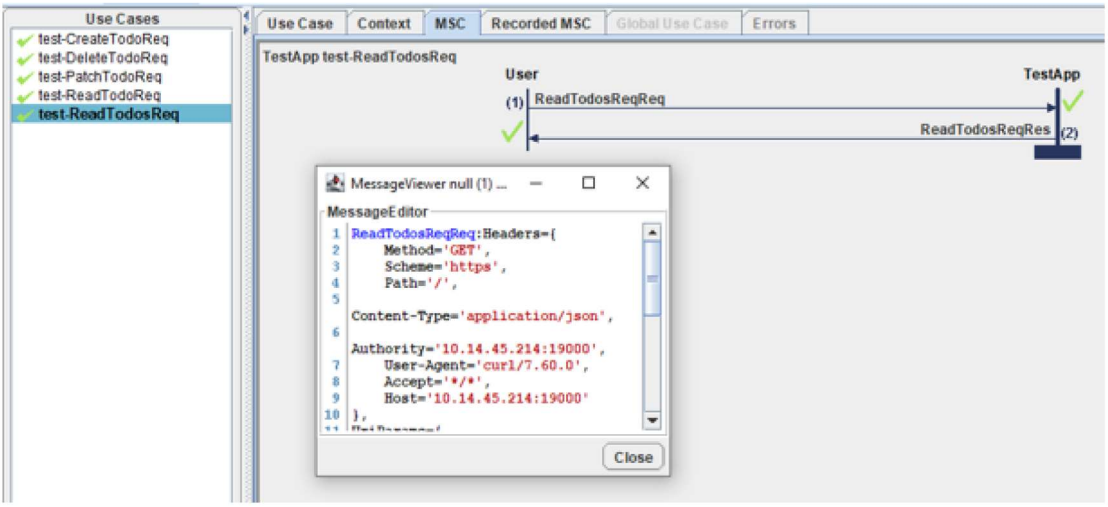
```
                /myRes = getRes(/Request);
                /myRes/UriParams/HttpStatusCode = 200;
                /myRes/Body/handler = "h_ReadTodoReq";
                reply /myRes;
            }.
        }
        h_ReadTodosReq {
            JCode = {
                /myRes = getRes(/Request);
                /myRes/UriParams/HttpStatusCode = 200;
                /myRes/Body/handler = "h_ReadTodosReq";
                reply /myRes;
            }.
        }
    }
    Tests {
        test-CreateTodoReq {
            Kind = Test
            MSC = {
                ##User this
                User->this:CreateTodoReqReq:Headers={Method='POST',
                    Scheme='https',Path='/',Content-Type='application/json',
                    Authority='10.14.45.214:19000',User-Agent='curl/7.60.0',
                    Accept='*/*',Host='10.14.45.214:19000'},UriParams=
                    {Method='POST',Uri='/',Question='',Protocol='HTTP',
                    Version='2'},QueryParams={},PathParams={},Body={}
                User<-this:CreateTodoReqRes:UriParams={HttpStatusCode=201}
            }.
        }
        test-DeleteTodoReq {
            Kind = Test
            MSC = {
                ##User this
                User->this:DeleteTodoReqReq:Headers={Method='DELETE',
                    Scheme='https',Path='',Content-Type='application/json',
                    Authority='10.14.45.214:19000',User-Agent='curl/7.60.0',
                    Accept='*/*',Host='10.14.45.214:19000'},UriParams=
                    {Method='DELETE',Uri='',Question='',Protocol='HTTP',
                    Version='2'},QueryParams={},PathParams={},Body={}
                User<-this:DeleteTodoReqRes:UriParams={HttpStatusCode=200}
            }.
        }
        test-PatchTodoReq {
            Kind = Test
            MSC = {
                ##User this
                User->this:PatchTodoReqReq:Headers={Method='PATCH',
                    Scheme='https',Path='',Content-Type='application/json',
                    Authority='10.14.45.214:19000',User-Agent='curl/7.60.0',
                    Accept='*/*',Host='10.14.45.214:19000'},UriParams=
                    {Method='PATCH',Uri='',Question='',Protocol='HTTP',
```

Slika 3.12: MyserverApi.sample

```
                        version='2'},QueryParams={},PathParams={},Body={}
                    User<-this:DeleteTodoReqRes:UriParams={HttpStatusCode=200}
                }.
            }
            test-PatchTodoReq {
                Kind = Test
                MSC = {
                    ##User this
                    User->this:PatchTodoReqReq:Headers={Method='PATCH',
                        Scheme='https',Path='',Content-Type='application/json',
                        Authority='10.14.45.214:19000',User-Agent='curl/7.60.0',
                        Accept='*/*',Host='10.14.45.214:19000'},UriParams=
                        {Method='PATCH',Uri='',Question='',Protocol='HTTP',
                        Version='2'},QueryParams={},PathParams={},Body={}
                    User<-this:PatchTodoReqRes:UriParams={HttpStatusCode=200}
                }.
            }
            test-ReadTodoReq {
                Kind = Test
                MSC = {
                    ##User this
                    User->this:ReadTodoReqReq:Headers={Method='GET',
                        Scheme='https',Path='',Content-Type='application/json',
                        Authority='10.14.45.214:19000',User-Agent='curl/7.60.0',
                        Accept='*/*',Host='10.14.45.214:19000'},UriParams=
                        {Method='GET',Uri='',Question='',Protocol='HTTP',
                        Version='2'},QueryParams={},PathParams={},Body={}
                    User<-this:ReadTodoReqRes:UriParams={HttpStatusCode=200}
                }.
            }
            test-ReadTodosReq {
                Kind = Test
                MSC = {
                    ##User this
                    User->this:ReadTodosReqReq:Headers={Method='GET',
                        Scheme='https',Path='/',Content-Type='application/json',
                        Authority='10.14.45.214:19000',User-Agent='curl/7.60.0',
                        Accept='*/*',Host='10.14.45.214:19000'},UriParams=
                        {Method='GET',Uri='/',Question='',Protocol='HTTP',
                        Version='2'},QueryParams={},PathParams={},Body={}
                    User<-this:ReadTodosReqRes:UriParams={HttpStatusCode=200}
                }.
            }
        }
        FormatVersion = 2
    }
  }
}
```

Slika 3.13: MyserverApi.sample

Na sljedećim slikama također možemo vidjeti primjer kako izgleda MSC u Rules Gui. Rules Engine omogućuje dodavanje nove logike pravila ili izmjenu postojeće logike. Dijagram toka predviđene poslovne logike može se pretvoriti u takozvanu aplikaciju, gdje dijelovi koda mogu kontrolirati tijek poruke. Alat Rules GUI omogućuje uređivanje postojeće i stvaranje nove poslovne logike uz vizualnu podršku i izravnu povratnu informaciju.



Slika 3.14: Primjer msc u Rules GUI

4 | Zaključak

Tema ovog završnog rada je bila razvoj Swagger code generatora za JCode programski jezik, koji je razvijen unutar tvrtke ENEA. U radu smo se upoznali kao funkcionira Swagger. Njegova najvažnija značajka je kada napišemo specifikaciju, Swagger alatima možemo razviti API u više smjerova. Prvo smo pomoću Swagger Editora napisali specifikaciju za "todo" aplikaciju u YAML datoteci te izgenerirali python-flask server. Zatim smo koristili Swagger Codegen. Swagger Codegen služi za generiranje poslužitelja za vlastiti API, a jedino što je preostalo je implementirati logiku poslužitelja. Možemo zaključiti da Swagger ima širok spektar alata koji ubrzavaju proces razvoja API-ja, možemo automatski generirati dokumentaciju, testove te poslužitelj. Ovaj rad je rezultirao uspješnim generiranjem JCoda. Swager Codegen je generirao klijentske biblioteke za API na više od 40 jezika, a ovim radom smo dodali još jedan, što je i bio cilj ovog rada.

Literatura

[1] \https://www.techtarget.com/searchapparchitecture/definition/Swagger

[2] \https://en.wikipedia.org/wiki/Swagger_(software)

[3] \https://www.redhat.com/en/topics/api/what-is-a-rest-api

[4] \https://hr.wikipedia.org/wiki/HTTP

[5] \https://www.extrahop.com/resources/protocols/http/

[6] \https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.bib.irb.hr%2F835032%2Fdownload%2F835032.0069064305_1035_Cop_Julian.pdf&psig=AOvVaw0w0x5QEVBWTdKBH7vFyYqL&ust=1666255863216000&source=images&cd=vfe&ved=0CAoQjRxqFwoTCKCHwIP16_oCFQAAAAAdAAAAABAq

[7] \https://swagger.io/docs/specification/about//

[8] \https://swagger.io/docs/specification/about//

[9] \https://www.infoworld.com/article/3296360/what-is-the-jdk-introduction-to-the-java-development-kit.html

[10] \https://en.wikipedia.org/wiki/Java_virtual_machine

[11] \https://www.ibm.com/cloud/learn/jre

[12] \https://en.wikipedia.org/wiki/JAR_(file_format)

[13] \https://en.wikipedia.org/wiki/Message_sequence_chart

[14] \https://upload.wikimedia.org/wikipedia/commons/e/e8/Typical_MSC_%28Message_Sequence_Chart%29.png

[15] \https://swagger.io/docs/specification/basic-structure/

[16] \https://en.wikipedia.org/wiki/Mustache_(template_system)

[17] \https://docs.docker.com/compose/