УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Стефан Пијетловић

Jedan pristup objedinjavanju alata za razvoj programske podrške

МАСТЕР РАД

Нови Сад, (2015)

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:			
Идентификациони број, **ИБР**:			
Тип документације, **ТД**:		Монографска документација	
Тип записа, **ТЗ**:		Текстуални штампани материјал	
Врста рада, **ВР**:		Дипломски – мастер рад	
Аутор, **АУ**:		**Стефан Пијетловић**	
Ментор, **МН**:		**Проф. др Никола Теслић**	
Наслов рада, **НР**:		**Један проступ обједињавању алата за развој програмске подршке**	
Језик публикације, **ЈП**:		Српски / латиница	
Језик извода, **ЈИ**:		Српски	
Земља публиковања, **ЗП**:		Република Србија	
Уже географско подручје, **УГП**:		Војводина	
Година, **ГО**:		**2015.**	
Издавач, **ИЗ**:		Ауторски репринт	
Место и адреса, **МА**:		Нови Сад; трг Доситеја Обрадовића 6	
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)			
Научна област, **НО**:		Електротехника и рачунарство	
Научна дисциплина, **НД**:		Рачунарска техника	
Предметна одредница/Кqучне речи, **ПО**:		**Алати за развој програмске подршке, Клијент-послужилац архитектура**	
УДК			
Чува се, **ЧУ**:		У библиотеци Факултета техничких наука, Нови Сад	
Важна напомена, **ВН**:			
Извод, **ИЗ**:		**У раду је представљен један приступ обједињавању алата за развој програмске подршке у циљу једноставнијег, бржег и ефикаснијег развоја. Приказана је и имплементација система у програмском језику Python и представљени су резултати.**	
Датум прихватања теме, **ДП**:			
Датум одбране, **ДО**:			
Чланови комисије, **КО**:	Председник:	**Доц. др Иштван Пап**	
	Члан:	**Доц. др Никола Челановић**	Потпис ментора
	Члан, ментор:	**Проф. др Никола Теслић**	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		Monographic publication	
Type of record, **TR**:		Textual printed material	
Contents code, **CC**:		Master Thesis	
Author, **AU**:		**Stefan Pijetlović**	
Mentor, **MN**:		**Nikola Teslić, PhD**	
Title, **TI**:		**One approach to unifying software development tools**	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Republic of Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		**2015.**	
Publisher, **PB**:		Author's reprint	
Publication place, **PP**:		Novi Sad, Dositeja Obradovica sq. 6	
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)			
Scientific field, **SF**:		Electrical Engineering	
Scientific discipline, **SD**:		Computer Engineering, Engineering of Computer Based Systems	
Subject/Key words, **S/KW**:		**Client-server architecture, Software development tools**	
UC			
Holding data, **HD**:		The Library of Faculty of Technical Sciences, Novi Sad, Serbia	
Note, **N**:			
Abstract, **AB**:		**This paper presents an approach to the unifying of software development tools in order to make the development simpler, faster and more efficient. The paper includes the system impelemntation in the Python programming langugage as well as the results.**	
Accepted by the Scientific Board on, **ASB**:			
Defended on, **DE**:			
Defended Board, **DB**:	President:	**Ištvan Pap, PhD**	
	Member:	**Nikola Čelanović, PhD**	Menthor's sign
	Member, Mentor:	**Nikola Teslić, PhD**	

Zahvalnost

Zahvaljujem se akademskom mentoru prof. dr Nikoli Tesliću i tehničkom mentoru Petru Jovanoviću na savetima i ukazanoj pomoći pri realizaciji ovog rada.

Zahvaljujem se kolegama Slobanu Prljeviću, Danijelu Kneževiću i Branislavu Rankovu na stručnoj pomoći i praktičnim savetima tokom izrade rada.

Posebno se zahvaljujem prodici, prijateljima i ostalim kolegama koji su bili tu da pruže podršku tokom čitavog školovanja.

SADRŽAJ

SPISAK SLIKA

SPISAK TABELA

SKRAĆENICE

REST - *REpresentational State Transfer*, arhitektura programske podrške za pravljenje proširivih veb servisa

WSGI - *Web Server Gateway Interface*, univerzalna sprega za veb poslužioce i veb aplikacije za programski jezik Python

LDAP - *Lightweight Directory Access Protocol,* protokol na aplikacionom nivou za pristupanje i održavanje sistema sa distribuiranim direktorijumima

TCP - *Transmission Control Protocol,* protokol na transportnom nivou zadužen za kontrolu sesije koji ide zajedno sa internet protokolom

IP - *Internet Protocol,* najrasprostanjeniji internet protokol na mrežnom nivou

HTTP - *HyperText Transfer Protocol,* protokol na aplikacionom nivou za distribuirane hipermedia informacione sisteme

API - *Application Program Interface,* aplikaciona programska sprega

XML - *Extensible Markup Language,* opisni jezik čitljiv i za ljude i za mašine

JSON - *JavaScript Object Notation,* format koji koristi tekst čitljiv ljudima za prenos objekata koji se sastoje iz parova atribut-vrednost

HTML - *HyperText Markup Langugage,* opisni jezik koji se koristi za izradu veb stranica

RFC - *Requiest For Comments,* memorandum internet inženjerske operativne grupe o standardima i protokolima vezanim za internet

ASN.1 - *Abstract Syntax Notation One,* standard i notacija za opis pravila, struktura, prenosa, šifrovanja i dešifrovanja podataka u telekomunikacijama i internet mrežama

OSI - *Open Systems Interconnection,* konceptualni model koji karakteriše i standarizuje fukncije za komunikaciju telekomunikacinih ili računarskih sistema bez obzira na njihovu internu realizaciju

URL - *Uniform Resource Locator,* referenca na resurs sa veba koji specificira njegovu lokaciju u nekoj računarskoj mreži

YAPSY - *Yet Another Plugin SYstem,* sistem za pravljenje dodatnih programa za programski jezik Python

VPN - *Virtual Private Network***,** virtuelna privatna mreža koja predstavlja proširenje lokalne mreže tako što omogućuje korisnicima pristup mreži i sa računara van nje

1. Uvod

Pored razvojnih okruženja koja su podrazumevana a ponekad i neizostavna, programeri tokom svog radnog dana koriste i veliki broj pomoćnih alata u cilju boljeg, bržeg, kvalitetnijeg pa samim tim i uspešnijeg razvoja programske podrške. Razvojem programiranja kao inženjerske discipline i uključivanja računara u svakodnevnicu, vremenom su nastali veoma složeni poduhvati u raznim oblastima računarstva. Razvoj složene programske podrške ima dosta izazova pre svega zbog velikog broja modula kao i velikog broja ljudi koji istovremeno rade na istom projetku. Programeri obično imaju drugačije ideje za rešavanje istog problema i različit nivo stručnosti i iskustva što otežava razvoj i dovodi do neminovnog pojavljivanja grešaka i nepredvidivog ponašanja jednom kada se pojedinačni moduli spoje u sistem. Ipak, neophodno je da više ljudi paralelno radi na istom projektu zbog njihovog obima i vremena isporuke. Kako bi se broj grešaka sveo na minimum a vreme isporuke maksimalno skratilo, koriste se razne tehnike kao što su praćenje verzija programske podrške, praćenje grešaka, analiza koda od strane drugih programera i mnoge druge. Sve ove tehnike imaju odgovarajuće alate uz pomoć kojih se sprovode.

U zavisnosti od oblasti kojom se bave, broj alata koje programeri koriste na dnevnom nivou može varirati od nekoliko do nekoliko desetina. Za efikasno rukovanje svakim od ovih alata potrebna je odgovarajuća obuka koja se takođe razlikuje od alata do alata i može trajati od par dana do par nedelja u zavisnosti od njihove kompleksnosti. U velikim programerskim kompanijama i institutima koje imaju izuzetno veliki broj zaposlenih, organizovanje obuke za veliki broj ljudi često nije jednostavno, jer se zaposleni ponekad nazale na geografski veoma udaljenim lokacijama kao što su različiti kontinenti, pa je problem između ostalog i vremenska zona. Ukoliko dođe do promene alata za praćenje verzija programske podrške, potrebno je

ponovo sprovesti obuku za drugi alat koji u suštini ima istu funkciju ali radi na drugačijim principima koji najčešće nisu od suštinske važnosti za programere.

Tema ovog rada je jedan pristup objedinjavanja alata za razvoj programske podrške u cilju pojednostavljenja obuke prilikom upoznavanja sa novim alatima, rešavanja problema raznolikosti velikog broja alata sa istom namenom, kao i pojednostavljenje prelaska sa jednog alata na drugi.

Vodeći se principom apstrakcije, programeri ne rukuju direktno samim programskim alatima već umesto toga koriste jednostavnu klijentsku konzolnu aplikaciju. U okviru same aplikacije je okruženje podešeno tako da programeru pruži samo ono od informacija što mu je potrebno za rad. Programeri u konzoli izdaju naredbe koje se potom obrađuju i u zavisnosti od komande prosleđuju poslužiocu preko REST sprege. Prozivanjem odgovarajuće metode poslužioca se potom proziva i odgovarajuća metoda nekog od alata koji su realizovani kao veb servisi a korisnik biva obavešten o uspešnosti celokupne operacije.

U slučaju da kompanija odluči da pređe sa jednog na drugi alat, programer uopšte ne mora ni da zna da je do promene došlo. Na ovakav način dobijamo uštedu na vremenu potrebnom za obuku kao i znatno jednostavniju proceduru što ima za posledicu veću produktivnost programera u toku radnog dana.

Klijentska aplikacija i poslužilac su realizovani u programskom jeziku Python. Korišćeno je Flask okruženje za razvoj veb aplikacija uz Yapsy sistem za dodatne module kako bi rešenje bilo proširivo, Argparse biblioteka za realizaciju jednostavnih konzolnih aplikacija kao i baza podataka čija je namena da skladišti zahteve upućene servisima i dostavi ih u slučaju da servisi nisu dostupni. U radu će biti prikazani primeri sa alatima otvorenog koda kao što je Bugzilla, alat za praćenje grešaka u programskoj podršci, kao i rukovanje imenikom uz pomoć LDAP protokola.

1.1 Organizacija rada

Ovaj rad je podeljen u 7 poglavlja.

Poglavlje broj 2 (Teorijske osnove) opisuje ukratko programski jezik Python, Flask, Argparse modul, Yapsy sistem za dodatne module, LDAP protokol i REST paradigmu čije je poznavanje potrebno za razumevanje rada. Takođe je dat kratak opis servisa otvorenog koda Bugzilla koji je korišćen u rešenju kao i tipičan proces praćenja i otklanjanja grešaka u programskoj podršci.

U trećem poglavlju (Koncept rešenja) se nalazi teorijski opis rešenja, opis arhitekture i razlozi zašto je arhitektura priklada za rešenje problema i dat je opis toka podataka u sistemu.

Četvrto poglavlje (Programsko rešenje) sadrži konkretnu realizaciju odnosno opis nekoliko najbitnijih klasa poslužioca i klijentske aplikacije.

Peto poglavlje (Ispitivanje i evaluacija) predstavlja rezultate rada.

U šestom poglavlju se nalazi zaključak rada i navedeni su mogući dalji pravci razvoja sistema.

Poslednje, sedmo, poglavlje sadrži literaturu korišćenu u izradi ovog rada.

2. Teorijske osnove

2.1 Python

Kao što je napomenuto u uvodu, u rešenju zadatka korišćen je programski jezik Python. Python je nastao početkom devesetih godina, konkretno 1991. godine, mada ideja za njegov nastanak datira iz osamdesetih. On je opštenamenski programski jezik visokog nivoa čiji dizajn podstiče pre svega čitljivost koda. To podrazumeva mogućnost da se programeri izraze koristeći manji broj linija nego što bi to bilo moguće uz pomoć nekih drugih široko rasprostranjenih programskih jezika kao što su C++ ili Java. Python podržava više programerskih paradigmi kao što su pored objektno orijentisanog i imperativnog programiranja još i funkcionalno programiranje, dinamičke tipove kao i automatsko rukovanje memorijom (programer ne mora voditi računa o oslobađanju memorije kada je jednom zauzme). Sve ove mogućnosti su doprinele da Python danas bude u deset najpopularnijih programskih jezika prema TIOBE indeksu [1].

Srž filozofije Python-a je sumirana u dokumentu nazvanom PEP 20 [2] i uključuje razne šaljive aforizme kao što su:

- Lepo je bolje nego ružno
- Eksplicitno je bolje nego implicitno
- Jednostavno je bolje nego kompleksno
- Kompleksno je bolje nego komplikovano
- Posebni slučajevi nisu dovoljno posebni da se pravila prekrše

Primećuje se da svi aforizmi stavljaju akcenat na čitljivost koda. Python je dizajnijran sa relativno malim jezgrom ali velikom standardnom bibliotekom i lako prošitivim interpreterom. Ideja koja stoji iza te odluke je da u Python-u „treba da postoji jedan poprilično očigledan način

kako rešiti neki problem“ (takođe jedan od aforizama). Drugi programski jezici imaju kompleksniju i širu sintaksu koja omogućava više načina za realizaciju iste ideje i samim tim potencijalno dovodi do nekonzistentnosti unutar koda ukoliko se ne prate neke konvencije ili stilovi kodiranja. Ovo stvara i više mogućnosti za nastajanje grešaka i nepredvidivog ponašanja. Još jedna pogodnost izražene čitljivosti Python-a je ta što je kod koji je čitljiv lakši za održavanje i samim tim se na duže staze doprinosi uštedi u vremenu.

2.1.1 Flask

Flask je mikro okruženje za razvoj veb aplikacija napisano u programskom jeziku Python koje se koristi i u svetski poznatim aplikacijama kao što je *LinkedIn*. Zasnovan je na *Werkzeug* WSGI biblioteci i *Jinja2* alatu za rukovanje šablonima. Iako nema ugrađen sloj za apstrakciju baze podataka ili proveravanje ispravnosti formi, Flask podržava i podstiče proširenja koja mogu jednostavno dodati razne funkcionalnosti u samo okruženje. Zbog toga je nazvan mikro okruženjem jer ne vrši pritisak na programera da koristi neke specifične alate, baze podataka, biblioteke ili poslužioce već pruža slobodu da programeri koriste ono što im odgovara uz mogućnost promene dizajna u svakom trenutku [3].

U trenutku kada je ovaj rad bio na samom početku, nije bila poznata celokupna arhitektura sistema. Nepoznati su bili poslužioci kao i baze podataka koje će biti korišćene. S obzirom na fleksibilnost koju pruža Flask, izabran je kao glavno okruženje korišćeno za razvoj poslužioca. Poznato je bilo da rešenje treba da ima veliki broj proširenja u vidu mnogobrojnih servisa i da omogući jednostavno dodavanje novih bez promena u celokupnom sistemu zbog čeka je korišćen i Yapsy sistem za dodatne module.

2.1.2 Argparse

Argparse je modul za pravljenje aplikacija koje koriste komandnu liniju u Python-u. Uz pomoć njega je moguće napraviti jednostavne konzolne aplikacije koje su prilagođene i korisnicima koji nisu naviknuti na rad sa komandnom linijom i korisnicima koji rukuju terminalom svakodnevno. Argparse definiše nekoliko klasa koje omogućavaju parsiranje korisničkih komandi i njihovu obradu dalje u kodu. U sam Argparse je ugrađena funkcionalnost u vidu pomoći kako je program potrebno pozvati i pravilno koristiti. Ovi razlozi su bili više nego dovoljni da Argparse bude glavni kandidat za realizaciju klijentske aplikacije [4].

Kako je ideja koja stoji iza rada apstrakovati rukovanje složenim alatima, aplikacija treba da bude što jednostavnija i intuitivnija. S obzirom da je okruženje podešeno tako da programer odmah ima dostupne sve informacije koje su mu potrebne, pokretanjem programa uz argument ---help ili --h, programer vidi koji su sve servisi njemu dostupni, kao i kako da ih pozove. Ukoliko korisnik pokrene program bez pozivanja ikakvih argumenata, dobiće poruku da prilikom poziva mora dodati neki od navedenih obaveznih argumenata i da doda argumente --h ili --help za više informacija. Ako objašnjenje za neki servis nije dovoljno detaljno, moguće je pozvati i pomoć nad svakim pojedinačnim servisom, gde je objašnjena svaka od komandi. Analogno, programer ima pregled svih komandi nekog servisa kao i koji su argumenti svake komande, i još podršku na nivou svake pojedinačne komande i njenih argumenata. Ovo ima pozitivan uticaj na brzo učenje i privikavanje na aplikaciju.

2.1.3 Yapsy

Yapsy je anagram za *Yet Another Plugin SYstem* koji doslovno preveden sa engleskog glasi "Još jedan sistem za dodatne module". Njegova glavna svrha je da omogući način da se jednostavno dizajnira sistem za dodatne module u Python-u koji će biti zavisni samo od njegove standardne biblioteke kao što je i sam Yapsy.

U svojoj osnovi, Yapsy definiše dve klase koje čine jezgro sistema: potpuno funkcionalnu i veoma jednostavnu klasu koja predstavlja jezgro sistema za rukovanje dodatnim modulima nazvana *PluginManager* i spregu *IPlugin* koja definiše API svih modula prema *PluginManager*-u. Ove klase su podešene tako da je samo u par linija koda moguće napraviti sistem koji će lako učitavati i aktivirati sve dodatne module. Potrebno je samo da svaki dodadni modul, odnosno u našem slučaju servis, uz sebe ima i odgovarajući opis u vidu datoteke sa ekstenzijom .yapsy-plugin i da se svi servisi nalaze na odgovarajućim mestima u hijerarhiji direktorijuma [5].

2.2 REST

REpresentational State Transfer ili skraćeno REST je stil za opis arhitekture računarskih sistema koji stavlja akcenat na komponente sistema i njihovu interakciju umesto na detalje implementacije i sintakse protokola. Cilj ovakve arhitekture jeste svesti kašnjenje i nepotrebnu komunikaciju u mreži na minimum a istovremeno težiti ka što većoj nezavisnosti komponenti i skalabilnosti celokupnog sistema. REST paradigma se sastoji od pet formalnih ograničenja koja oblikuju sistem da podseća na i ponaša se kao dobro dizajnirana veb aplikacija [6].

. Ova ograničenja su sledeća:

- Klijent-poslužilac arhitektura
- Komunikacija bez čuvanja stanja
- Odgovori se „keširaju“ odnosno čuvaju privremeno
- Sistem treba da ima slojevitu strukturu
- Sistem treba da ima uniformnu spregu

Postoji još opcionih ograničenja koja nisu fundamentalna i koja nisu podržana u našoj impelementaciji tako da neće biti navedena i detaljno opisana.

Postojanje klijent-poslužilac arhitekture podrazumeva razdvajanje zadataka – svaka strana se koncentriše samo na ispunjavanje svog dela što čini implementaciju obe strane jednostavnijom. U našem slučaju, na klijentskoj strani se nalazi jednostavna logika za parsiranje komandi kao i vrlo jednostavna grafička sprega sa korisnikom, dok se kompleksnija logika rukovanja zahtevima nalazi na strani poslužioca. Ovakva arhitektura omogućava nezavisno razvijanje obe strane jednom kada se infrastruktura uspostavi i definiše API. Promene na strani poslužioca nemaju uticaja na klijente ukoliko nije došlo do promene samog API-a jer klijenti nemaju uvid u to šta se dešava „ispod haube“. Ovo takođe stvara mogućnost više vrsta klijentskih aplikacija, od kojih neke mogu biti konzolne a neke mogu uključivati napredniju grafičku korisničku spregu.

Komunikacija bez čuvanja stanja znači da u toku jedne sesije, teret očuvanja virtuelnog stanja aplikacije spada na klijenta. Poslužilac nigde ne čuva stanje vezano za klijente pa je samim tim jednostavniji za implementaciju i omogućava rukovanje većim brojem klijenata istovremeno. Umesto toga, svaki zahtev upućen od strane klijenta je nezavisan i svaki odgovor poslužioca sadrži sve neophodne informacije kako bi bilo moguće napraviti novi zahtev i tako preći u naredno virtuelno stanje aplikacije. Posledica je da nema potrebe za poznavanjem celokupne topologije sistema.

Keširanje odgovora je prisutno radi veće efikasnosti samog sistema. Na ovaj način privremenog čuvanja odgovora se znatno umanjuje a u nekim slučajevima i potpuno izbegava nepotrebna komunikacija između klijenta i poslužioca. Ukoliko se zahtevani resursi nisu promenili između dva uzastopna zahteva, nema potrebe da se klijent obrati poslužiocu već može iskoristiti svoju kopiju resursa. Ovim se smanjuje ne samo opterećenje na mreži i na strani poslužioca već se postiže i bolji odziv sa klijentske strane pošto nema nepotrebnog čekanja odgovora.

Slojevita struktura sistema podrazumeva postojanje posrednika i svaki sloj vidi samo svoje prve „komšije“ odnosno slojeve sa kojima direktno komunicira. Ovakav vid pipe-and-filter

strukture odgovara našem sistemu jer povećava skalabilnost i omogućava bolje preraspoređivanje opterećenja na sistem [7].

Uniformna sprega predstavlja osnovu dizajna svakog RESTful sistema. Ovo fundamentalno ograničenje ima četiri principa: identifikacija resursa, manipulacija resursima, samoobjašnjavajuće poruke i hipermedija kao pokretač stanja aplikacije (*hypermedia as the engine of application state* ili skraćeno HATEOAS).

Identifikacija resursa podstiče ideju da se resursi koncetualno razlikuju od svoje reprezentacije koja se šalje klijentima kao odgovor. Primera radi, ukoliko je zahtevani resurs pravougaonik, njegova reprezentacija može biti dva realna broja koja predstavljaju dužinu njegovih stranica. Drugačija reprezentacija ovog istog pravougaonika može biti četiri para ili uređene trojke brojeva gde svaki par/trojka predstavlja teme pravougaonika u nekom koordinatnom sistemu. Ove reprezentacije iako različite, daju dovoljno informacija da se konstruiše željeni resurs u potpunosti, odnosno moguće je odrediti njegove fundamentalne osobine kao što su obim i površina tog pravougaonika. Ovakva reprezentacija se potom šalje klijentima u nekom od standarnih formata (XML, JSON, HTML...).

Manipulacija resursima znači da jednom kada klijent dobije reprezentaciju određenog resursa, ima sve informacije potrebene da taj resurs modifikuje ili obriše sa servera ukoliko za to ima dozvolu. U našem slučaju, klijent može od poslužioca zatražiti listu svih grešaka na kojima programer trenutno radi. Kada jednom dobije informacije o greškama, programer može da ih označi kao da su rešene i time menja resurs odnosno njegovo stanje ili ga uklanja iz sistema.

Princip samoobjašnjavajućih poruka cilja na to da svaka poruka koja se razmeni treba da sadrži dovoljno informacija o tome kako će biti obrađena, na primer koju vrstu parsera je potrebno iskoristiti.

Ideja iza HATEOAS-a je da se klijent navigira kroz aplikaciju koristeći hiperlinkove kako bi prešao u naredno stanje aplikacije. Pri tom, klijent podrazmeva da ne postoje druga stanja osim onih koja su definisana u odgovoru.

REST se oslanja na HTTP protokol i koristi najčešće četiri metode u zahtevima, a to su: GET, PUT, POST i DELETE. Uz to, koriste se i razne druge pogodnosti HTTP instrastrukture kao što su slojeviti proxy zastupnici i kešranje.

2.3 LDAP

LDAP je industrijski standardizovan protokol na aplikacionom nivou za pristupanje i održavanje sistema sa distribuiranim direktorijumima preko IP-a. Specificiran je od strane internet inženjerske operativne grupe u dokumentu RFC 4511. Opisan je uz pomoć apstrakntog

jezika za opis sintakse ASN.1. Nastao je devedesetih godina kao alternativa složenijim protokolima za rukovanje direktorijumima preko mreže jer za LDAP nije bio potreban ceo OSI protokol stek već jednostavniji i danas široko rasprostranjeni TCP/IP protokol stek.

Klijent započinje sesiju tako što se poveže sa LDAP poslužiocem, koji se još naziva i sistemski agent za direktorijume (DSA – Directory System Agent), preko standardnog porta za TCP. Nakon uspešnog povezivanja, klijent potom šalje zahteve bez potrebe da sačeka odgovore između dva uzastopna zahteva dok poslužilac šalje odgovore u proizvoljnom redosledu. Neke od operacija koje klijent može zahtevati su: dodavanje novih stavki, njihova izmena i brisanje, pretraga, povezivanje uz autentifikaciju i mnoge druge.

Servisi koji rade sa direktorijumima su veoma važni za razmenjivanje informacija o korisnicima, sistemima, servisima i aplikacijama unutar mreže pa i o samoj mreži. Primeri ovakvih servisa su e-mail direktorijum unutar neke organizacije ili telefonski imenik. Tipičan upit koji se može izvršiti nad ovakvim servisima je izlistavanje e-mail adresa i telefona svih programera koji rade na nekom projektu.

2.4 Bugzilla i praćenje/otklanjanje grešaka u sistemu

Bugzilla je alat otvorenog koda za praćenje grešaka u programskoj podršci, nastao inicijalno u okviru Mozilla projekta. Vremenom je usvojen i od strane drugih velikih organizacija kao što su WebKit, Linux kernel, Eclipse i mnogi drugi. Sve verzije počevši od 2.0 su napisane u programskom jeziku Perl. Iako ima potencijal da postane alat za celokupnu podršku razvoja projekata koja uključuje i rukovanje zadacima programera, akcenat pri razvoju Bugzilla-e je ostao samo na praćenju grešaka u sistemu na šta i samo ime implicira [8].

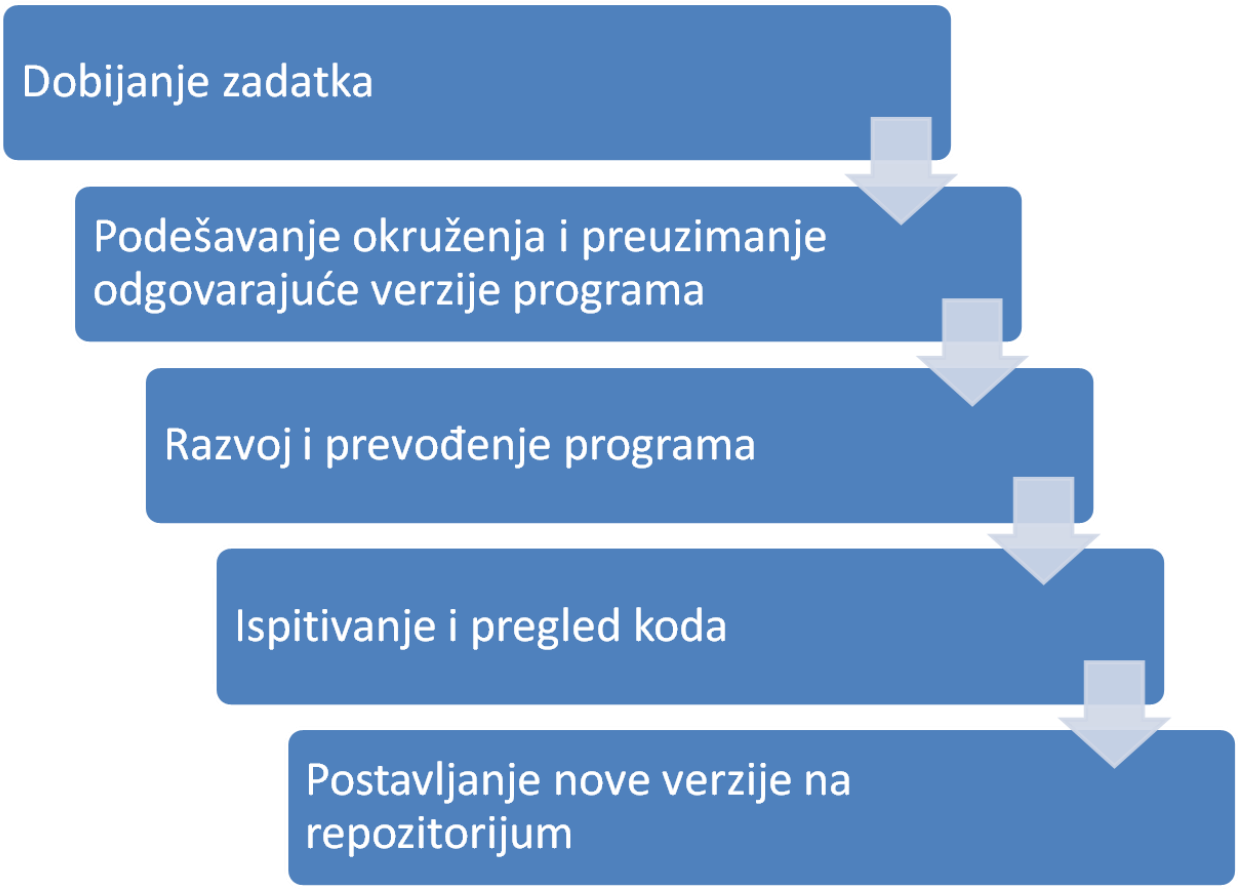
Praćenje grešaka je danas neizostavan proces pri razvoju iole kvalitetne programske podrške jer ne postoji primerak koji u sebi nema greške. Ukoliko program zaista ne ispoljava nikakva nepredviđena ponašanja i zadovoljava specifikaciju, onda samo greške nisu još uvek uočene ili se nisu ispoljile. Uticaj grešaka na ponašanje i odziv celog sistema se može razlikovati od jedva primetnih u vidu vrlo kratkog zastoja u radu koje se pojavlju vrlo retko i čije je ponašanje teško reprodukovati do veoma ozbiljnih koje čine program neupotrebljivim ili koje mogu dovesti do velike materijalne štete pa i gubitka ljudskih života. Najveći broj grešaka nastaje u samom kodu ili dizajnu programske podrške, bilo nemarom programera ili jednostavno nemogućnošću da se predvide svi mogući scenariji korišćenja programa. Ređe su greške koje nastaju usled neusaglašenosti razvojnih okruženja i operativnih sistema sa programom ili one

koje nastaju tako što programski prevodioci proizvedu neispravan kod zbog optimizacija ili nekih drugih razloga. Zato je prilikom opisa neke greške potrebno dostaviti što više informacija – pod kojim uslovima dolazi do greške, gde se ona ispoljava, koja je verzija programa, koji operativni sistem i platforma se koriste itd.

Sve ovo je prouzrovokalno da se velika količina truda od strane industrije uloži u prevenciju i pronalaženje načina za efikasno otklanjanje grešaka. Neke od tehnika za prevenciju nastajanja grešaka su rad na stilu programiranja kako bi greške bile lakše uočljivije i odmah primećene od strane prevodioca, automatizovano ispitivanje gde program sam pokreće testove i javlja programeru da li je do greške došlo, podsticaj programera da pišu jednostavne naredbe i čitljive programe, analiza koda od strane drugih programera...S druge strane, kada do greške već dođe sledi naporan zadatak u vidu pronalaženja porekla greške i njenog ispravljanja. Stvari koje olakšavaju posao programeru su program za otklanjanje grešaka (*debugger*) koji pruža bolji uvid u izvršavanje programa uz pomoć kontrolisanog izvršavanja korak po korak odnosno naredbu po naredbu kao i neometano izvršavanje programa do tačke prekida. Takođe neretko programer dobija i uvid u stanje programa kao što su vrednosti promenljivih i memorije. Kada program nema *debugger* koriste se raznorazni kontrolni ispisi bilo na ekran ili u datoteku dnevnika kako bi se suzio prostor koji treba pretražiti.

3. Kocept rešenja

Na slici je dat primer kako izgleda rešavanje jednog zadatka ili jedne greške u programskoj podršci unutar neke kompanije.

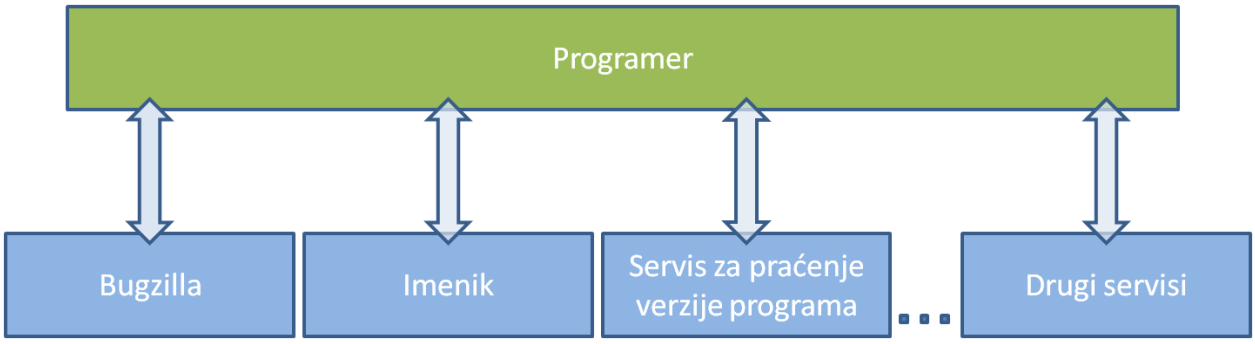


Slika 1 – Tok rešavanja programerskog zadatka

Nakon što programeru bude dodeljen zadatak koji treba da uradi, prva stvar koja sledi jeste podešavanje odgovarajućeg radnog okruženja. Potrebno je preuzeti trenutnu verziju programa i

eventualno uključiti sve neophodne dodatne alate. Potom sledi razvoj ili otklanjanje greške i ponovno prevođenje programa. U zavisnosti od obima programa, prevođenje se može raditi lokalno na računaru na kome se razvija ili na nekom poslužiocu koji ima znatno veću procesnu moć ili u nekim slučajevima na više poslužioca istovremeno ukoliko je program izuzetno veliki. Zatim ide ispitivanje koje može biti automatizovano u vidu unit-testova ili ručno i neretko neki vid pregleda koda od strane iskusnijih kolega. Tek kada program prođe uspešno sve provere i neko da potvrdu da je kod u redu, sledi postavljanje nove verzije programa na repozitorijum. U zavisnosti od metodologije razvoja, neretko se koriste i alati na kojima se na dnevnom nivou beleži koliko je procentualno zadatka urađeno kako bi grupovođe imale bolji uvid o napretku projekta.

Za nekoga ko je već iskusan u poslu, rukovanje za nekoliko alata u isto vreme ne predstavlja problem, ali svakako da se određeni deo vremena ne koristi optimalno pošto je nekada potrebno iste ili slične podatke uneti na više mesta. Međutim ukoliko kompanija ne može da postigne da uradi sav posao na vreme, neretko se unajmljuju programeri ili celi programerski timovi iz drugih kompanija da urade taj posao ili deo posla za njih, takozvano *outsource*-ovanje. U takvom slučaju je potrebno izvršiti obuku tog tima ljudi što je pogotovo nezgodno ukoliko su svi alati koji se koriste prilikom razvoja razvijeni unutar same kompanije. a tim unajmljenih ljudi nije u stanju da fizički prisustvuje obuci, što zbog nepostojanja odgovarajućeg prostora ili jednostavno velike geografske udaljenosti u slučajevima kada se posao prebacuje u druge države.

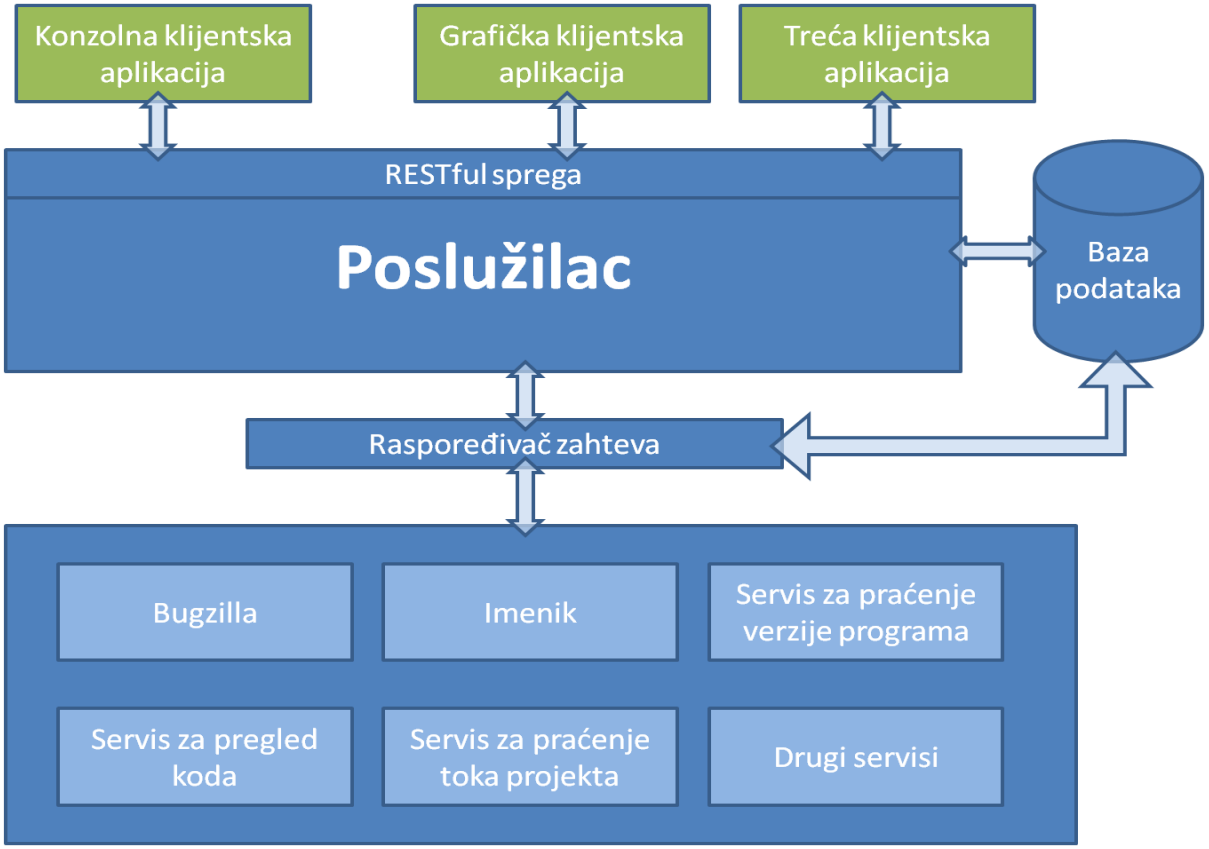


Slika 2 – Uobičajeno korišćenje servisa

Arhitektura našeg sistema je veoma prikladna za rešavanje ovakvog problema jer umesto obuke nad nekolicinom alata, dovoljno je obučiti unajmljene programere da koriste samo jedan – konzolnu klijentsku aplikaciju.

Na slici 3 se vidi da je ahritektura podeljena u 3 celine:

- Klijentske aplikacije
- Poslužilac
- Veb servisi



Slika 3 – Arhitektura sistema

Kao što se primeti, na jednom kraju sistema imamo klijentske aplikacije koje mogu biti konzolne, realizovane u veb pretraživaču ili kao potpuno zasebni programi/delovi nekog okruženja za razvoj. Poslužilac služi kao posrednik između svih komponenti. Veb servisi se nalaze nasuprot klijentskih aplikacija i pružaju razne funkcionalnosti koje su neophodne pri razvoju programske podrške.

Tipčan slučaj korišćenja sistema jeste da programer preko aplikacije zatraži neku od standardnih funkcionalnosti nekog od alata. Ne mora ni biti svestan koji je alat u pitanju, bitne su samo funkcionalnosti. Bez izuzimanja opštosti, u radu će uvek biti adresiran servis Bugzilla.

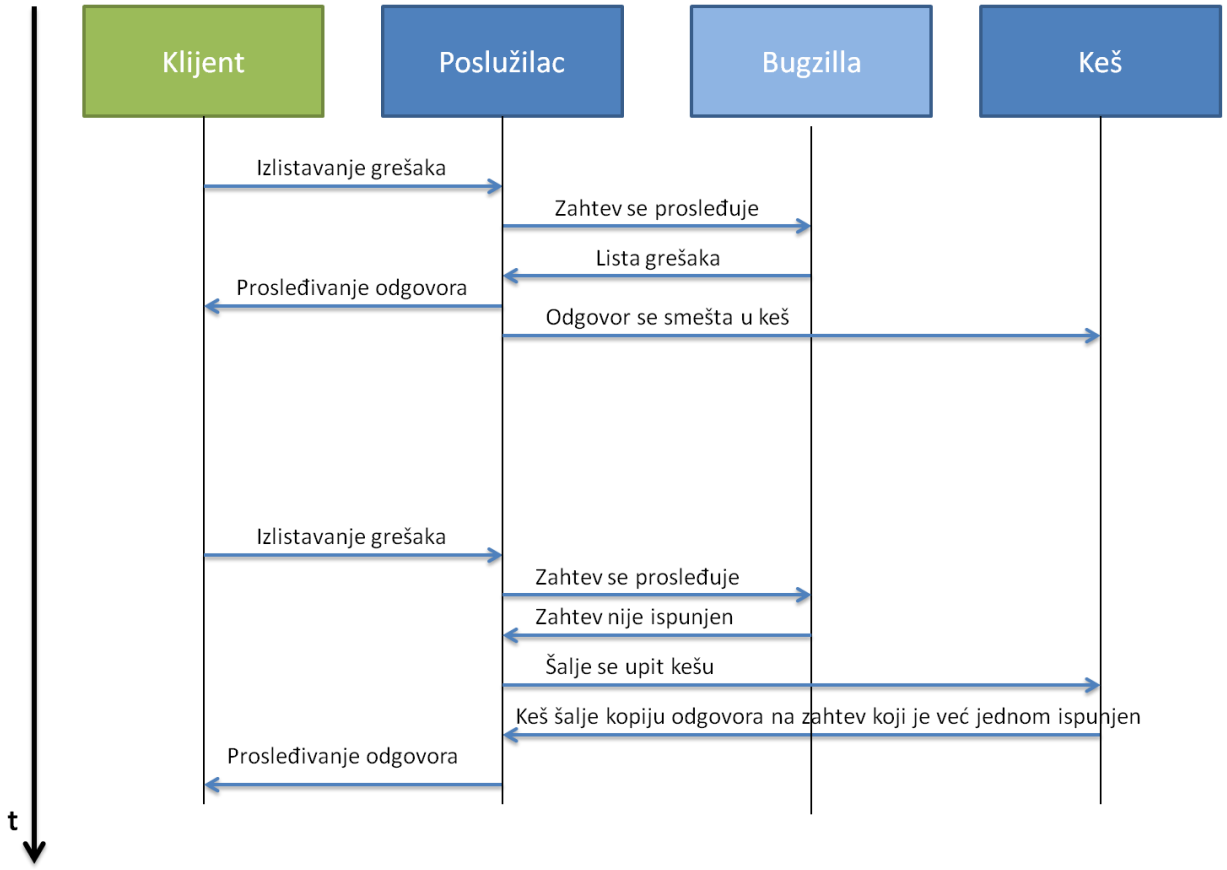
3.1 Poslužilac

Poslužilac predstavlja jezgro celokupnog sistema i zadužen je za komunikaciju sa svim preostalim delovima sistema. Sa jedne strane, komunicira sa klijentima putem REST sprege. Svaki servis ima po jednu REST metodu na poslužiocu za svaku funkcionalnost koju obezbeđuje. Tako na primer ukoliko je u pitanju veb servis Bugzilla, a funkcionalnosti koje želimo da pokrijemo su: izlistavanje svih grešaka na kojima radi programer, prijavljivanje novih

grešaka, ažuriranje informacija o postojećim ili dobavljanje detaljnih informacija o nekoj posebnoj grešci, na poslužiocu bi imali četiri metode sa vrlo sličnim putanjama:

- adresa_poslužioca/bugzilla/list
- adresa_ poslužioca /bugzilla/new
- adresa_ poslužioca /bugzilla/update
- adresa_ poslužioca /bugzilla/bug

Za neke od ovih metoda postoje i odgovarajući upitni parametri kao što su primera radi naziv nove greške koju želimo da prijavimo, identifikator greške koju želimo da ažuriramo ili čije informacije želimo detaljno da vidimo.



Slika 4 – Tok podataka u slučaju da je servis dostupan i nedostupan

Sa druge strane, preko neobaveznog raspoređivača zahteva, poslužilac komunicira sa veb servisima, prosleđivajući sve neophodne parametre iz klijentske aplikacije. U zavisnosti od samog veb servisa i zahtevi koji se formiraju su veoma različiti.

Takođe sistem sadrži lokalnu bazu podataka u kojoj se čuvaju zahtevi koji su upućeni ka veb servisima. U slučaju da od veb servisa nije moguće dobiti odgovor, bilo zbog preopterećenosti ili jednostano otkaza, klijentu se prosleđuje kopija odgovora. Na ovaj način

ukoliko nije došlo do promene stanja, klijent i dalje može da raspolaže sa podacima koje u slučaju da ne postoji keš ne bi mogao da dobije.

3.2 Raspoređivač zahteva

Raspoređivanje zahteva je je tehnika koja se koristi za maksimalno iskorišćavanje resursa, vremena u cilju povećanja protoka korisnih informacija. Glavna uloga ove komponente u našem sistemu je da omogući da svi zahtevi na kraju stignu do željenih servisa. U slučaju da preveliki broj zahteva stigne istovremeno do nekog servisa on neće biti u stanju da na sve odgovori u dogledno vreme i može doći do gubitka zahteva i prekida sesije.

Raspoređivač zahteva vodi računa o dve stvari: prva je da servisi ne dođu u stanje preopterećenosti a druga jeste sigurnija isporuka odgovora.

U prvom slučaju, kada imamo prevelik broj zahteva, raspoređivač privremeno čuva zahteve i ne prosleđuje ih istog trenutka ka servisima. Na ovaj način umesto da u jednom trenutku stigne preveliki broj zahteva i dođe do uskog grla, zahtevi se vremenski raspoređuju na veći interval. Ušteda je u tome što umesto da klijent sačeka neko vreme samo da bi dobio odgovor da je konekcija neuspešna i da šalje novi zahtev, odgovor će dobiti sa manjim zakašnjenjem ali bez ponovnog slanja zahteva. Na ovaj način se delimično reguliše saobraćaj i smanjuje nepotrebna komunikacija unutar mreže.

Drugi slučaj koji pokriva naš sistem je ukoliko servis nije u stanju da ispuni zahtev bilo da je to zbog prevelikog opterećenja ili zbog otkaza. U tom slučaju raspoređivač se obraća bazi podataka. Ukoliko je korisnik relativno skoro tražio neke informacije koje trenutno nisu dostupne, moguće je isporučiti kopiju ukoliko u međuvremenu nije došlo do promene stanja.

3.3 Baza podataka

Baza podataka našeg sistema služi za čuvanje zahteva upućenih prema servisima odnosno kao keš. Glavna namena ove komponente jeste da omogući isporuku informacija veb servisa u slučajevima kada sami veb servisi ne mogu da ih pruže kao što je u slučaju preopterećenosti ili prestanka rada.

Nakon svakog korisničkog zahteva prema nekom servisu, u bazu se smešta informacija koji korisnik je kontaktirao koji servis, koju metodu i koji odgovor je dobio. Kada se unesu nove

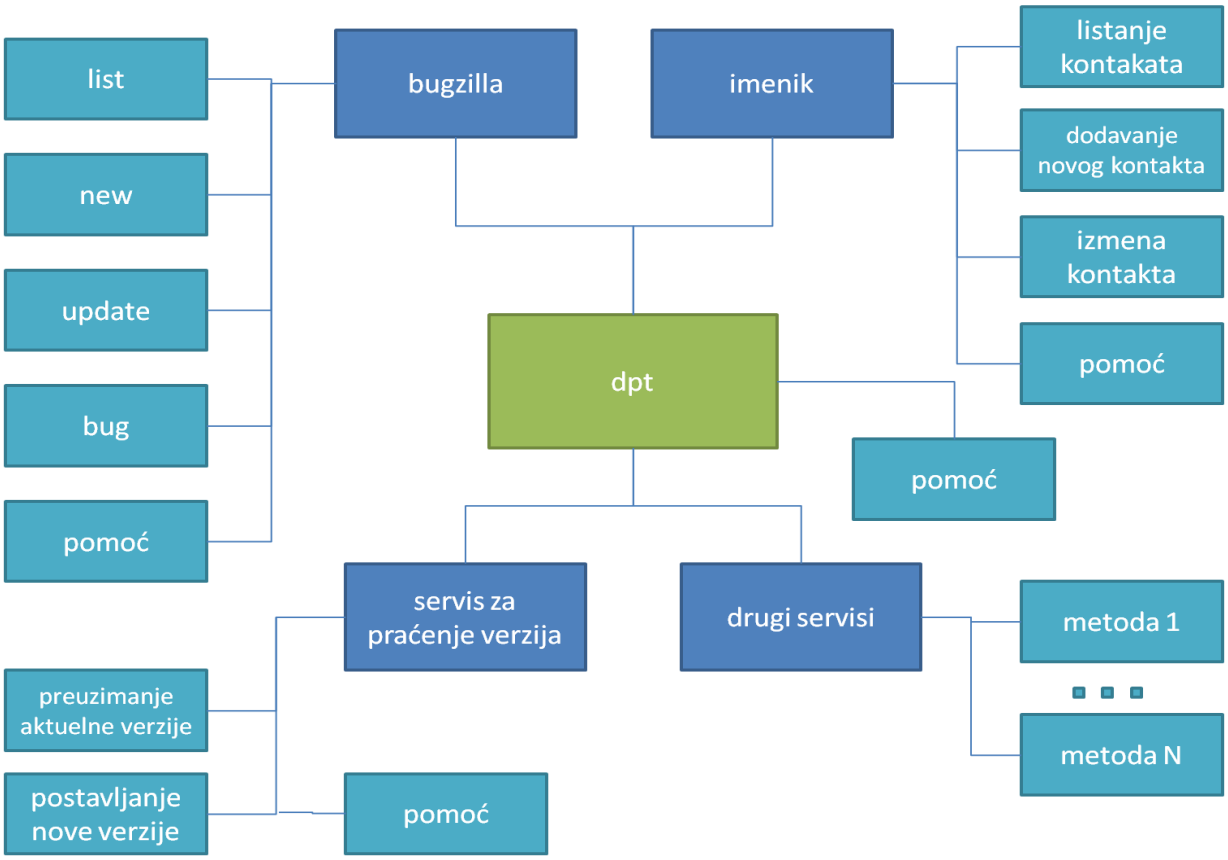
informacije u sistem odnosno kada korisnik drugi put pozove istu metodu, stari odgovor se zamenjuje novim. Na ovan način u bazi uvek imamo najnovije informacije.

Što se tiče informacija o servisima odnosno kojim servisima koji korisnik ima prava da pristupi se nalaze na drugom mestu. Svaki od servisa ima svoju bazu podataka sa svim neophodnim informacijama.

3.4 Klijentska aplikacija

Zadatak klijentske aplikacije jeste da omogući jednostavno i uniformno rukovanje sa svim alatima neophodnim za razvoj. Jednostavnost se ogleda u intuitivnosti same aplikacije kao i u preglednosti koju pruža. Sve komande se mogu predstaviti jednim stablom gde u korenu imamo glavnu komandu, nakon čega sledi ime željenog servisa i na kraju funkcionalnost.

Prilikom prvog susreta sa aplikacijom programer treba da pozove glavnu komandu *dpt* sa opcionim parametrom za pomoć. Ukoliko pozove sa neodgovarajućim argumentima, aplikacija će ga obavestiti o ispravnom načinu korišćenja. Takođe je dostupna i pomoć za svaki od servisa kao i za svaku od funkcionalnosti servisa.



Slika 5 – Stablo parsiranja komandi u klijentskoj aplikaciji

Primera radi, ukoliko je programer otklonio grešku koja mu je bila dodeljena i sada želi da to označi i na servisu Bugzilla, procedura je sledeća: ukoliko ne zna identifikator greške koju je otklonio, prvo treba da izlista sve greške koje su njegovo zaduženje. Komanda za izlistavanje glasi *dpt bugzilla list*. Aplikacija će sama pokupiti korisničko ime programera iz korisničkog direktorijuma i uputiti zahtev poslužiocu na URL *adresa_poslužioca/bugzilla/list* i korisničko ime ubaciti kao upitni parametar. Konačni URL bi potom izgledao ovako: *adresa_poslužioca/bugzilla/list?user=xy* gde xy predstavljaju korisničko ime programera. Kao rezultat će programer dobiti listu svih grešaka koje se vode na njegovo ime zajedno za informacijama o njima. Sledeći korak jeste označavanje greške kao rešene. Komanda bi u tom slučaju izgledala ovako: *dpt bugzilla resolve --bug identifikator_greške*. Analogno prvom slučaju se formira novi URL koji izgleda ovako:
adresa_poslužioca/bugzilla/resolve?user=xy&bug=ime_greške. Isti postupak važi i za sve preostale metode.

Drugi primer je korišćenje imenika. Tipičan imenik unutar jedne kompanije sadrži informacije poput imena zaposlenih, kontakt telefona (poslovnih i privatnih u nekim slučajevima), email adrese i druge kontakte, lokacija gde zaposleni radi (kancelarija, zgrada, grad, država i slično) kao i razne tehničke informacije vezane za tim u kome se zaposleni nalazi i tako dalje. Većina institucija koja ima iole veći broj zaposlenih sadrži neki vid imenika kako bi jedni zaposleni mogli jednostavno i brzo da kontaktiraju druge.

Česta je situacija da programer radi na grešci koju nije otkrio niti prijavio već je to odradio neko drugi. Nije redak slučaj da u opisu same greške nema dovoljno informacija o grešci niti kako je ponovo veštački proizvesti. Ovo ponekad može dovesti do privremene blokade u radu te je tada najbolje kontaktirati osobu koja je prijavila grešku i na taj način razrešiti sve nedoumice.

Kada korisnik uputi zahtev za detaljne informacije o nekoj grešci, dobija između ostalog i podatak ko je tu grešku objavio. Analogno komandama za izlistavanje grešaka, korisnik može dobiti i informacije o ostalim zaposlenima. Sve što treba da uradi jeste da u konzolnoj aplikaciji ukuca komandu *dpt directory info --user korisničko_ime*. Tada će se na ekranu prikazati svi podaci pomenuti u prethodnom pasusu naravno uz ograničenje da dato korisničko ime postoji, kao i da je poslužilac i veb servis u radnom stanju.

Alternativa je da korisnik sam ručno pretraži imenik koji je neterko realizovan kao neki veb servis ili zasebna aplikacija. To zahteva da korisnik prestane da radi ono što je u tom trenutku radio, pokrene aplikaciju ili pretraživač ukoliko nije već bio otvoren i onda obavi pretragu. Ovim se ne samo produžava vreme pretrage nego se i odvlači pažnja. Bolje je ukoliko

korisnik ne mora da napušta terminal i sve neophodne informacije za razvoj može da dobije preko par jednostavnih komandi.

Iz ova dva primera vidimo da sve komande imaju u suštini isti format. Koren svake komande jeste ključna reč *dpt* koja pokreće samu klijentsku aplikaciju. Potom sledi ime servisa čije usluge korisnik želi, praćeno sa odgovarajućom specifičnom uslugom. Ukoliko su za dobijanje ove usluge potrebe i dodatne informacije od strane korisnika one se navode po sistemu *--dodatni_parametar vrednost_parametra*. Ovo ima pozitivan efekat na veoma brzo i jednostavno prilagođavanje na aplikaciju.

4. Programsko rešenje

Kao što je već napomenuto ranije u radu, glavni deo rešenja (klijentska aplikacija i poslužilac) je implementiran u programskom jeziku Python, verzija 2.7. U poglavlju je dat pregled najbitnijih klasa poslužioca i klijentske aplikacije, kao i neki detalji realizacije klasa neophodnih za rukovanje servisom za praćenje grešaka..

4.1 Najbitnije klase poslužioca

Za realizaciju poslužioca je korišćen Flask, mikro okruženje za razvoj proširivih veb aplikacija. Kao što u dokumentaciji samog Flaska stoji – mirko ne znači da cela aplikacija mora da stane u jednu Python datoteku (iako je moguće) niti da je funkcionalnost koju ovo okruženje pruža nedovoljno. Ideja je da jezgro bude jednostavno i proširivo bez pritisaka na odluke poput koju bazu podataka je potrebno koristiti. I sve ove odluke se lako i brzo menjaju na licu mesta. Flask nudi sve što jednom programeru potrebno za razvoj jedne veb aplikacije i ne opterećuje ga ni sa kakvim dodatnim mogućnostima koje mu nisu potrebne. [3]

Flask ima mnogo raspoloživih konfiguracija i nekoliko konvencija koje treba pratiti. Neke od njih su da se šabloni i statične datoteke čuvaju u poddirektorijumima pod nazivom *templates* i *static*. Ovo je naravno moguće promeniti ali za to uglavnom nema razloga. Postoje i mnoge druge i kompleksnije konvencije ali za potrebe ovog rada su korišćene podrazumevane.

Uz Flask je korišćen i Yapsy, sistem za dodatne module u Python-u. On podržava jednostavno definisanje dodatnih modula i njihovu ugradnju u već postojeći sistem. Uz svaki dodatni program ili u našem slučaju uz svaki servis, ide i odgovarajuća yapsy-plugin datoteka

koja sadrži najosnovnije informacije o modulu, kao što su njegovo zvanično ime i interno koje će bii korišćeno unutar aplikacije.

4.1.1 Klasa Service_manager

Kao što joj i ime implicira, klasa *Service_manager* je zadužena za rukovanje svim veb servisima. Ova klasa vodi računa o servisima koji su trenutno uključeni u sistem. Najvažnije metode su:

- **get_service(name)**: metoda kao argument uzima ime servisa i ukoliko je servis registrovan vraća instancu klase tog sersvisa, u suprotnom, vraća vrednost *None*. Kada se vrednost *None* pojavi je dobra indikacija da nešto nije u redu sa servisom i da u slučaju da klijent zatraži neke njegove funkcionalnosti da je potrebno obratiti se i kešu

- **get_service_list()**: metoda bez argumenata koja vraća listu svih registrovanih servisa

- **update_services()**: metoda bez argumenata koja osvežava podatke u listi svih registrovanih servisa, time što proverava da li su se pojavili novi odnosno da li su postojeći svi na broju

4.1.2 Klasa Bugzilla

Klasa koja je zadužena za komunikaciju sa servisom Bugzilla. Ima nekoliko RESTful metoda koje su direkno zadužene za komunikaciju sa veb servisom. Zajedničko za sve metode je da je korisničko ime očitano iz radnog direktorijuma a svi dodatni parametri su preuzeti iz URL upitnih parametara. Uz klasu je neophodan i yapsy-plugin opis modula koji sadrži osnovne informacije o njemu. Izgled jednog trivijalnog yapsy-plugin opisa izgleda ovako:

```
[Core]
Name = Bugzilla Service
Module = bugzilla

[Documentation]
Author = Stefan Pijetlovic
Version = 0.1
Website = www.rt-rk.com
Description = Module that handles requests for bugzilla
```

Najbitnije metode ove klase su:

- **list_bugs()**: metoda koja izlistava sve greške na koima radi programer; nema dodatnih upitnih parametara. URL na koji je potrebno uputiti HTTP GET zahtev da bi se ova metoda prozvala je *adresa_poslužioca/bugzilla/list?user=username*.

- **info()**: metoda koja kao rezultat vraća detaljne informacije o datoj grešci; neophodan parametar jeste identifikator greške. URL na koji je potrebno uputiti HTTP GET zahtev da bi se ova metoda prozvala je:
adresa_poslužioca/bugzilla/info?user=username&bugid=identifikator_greške.

- **new()**: metoda za prijavljivanje nove greške; parametri koji su neophodni su ime greške kao i njen opis. URL na koji je potrebno uputiti HTTP POST zahtev da bi se ova metoda prozvala je *adresa_poslužioca/bugzilla/new?user=username&bug=ime_greške*. Opis greške se šalje kao tekstualni dokument i samim tim pošto dostavljamo nove podatke, moramo koristiti POST metodu.

- **update()**: metoda za ažuriranje informacija u vezi greške; parametri koji su neophondi su identifikator greške kao i opis greške u vidu tekstualnog dokumenta u kome se nalaze izmene. URL na koji je potrebno uputiti HTTP POST zahtev da bi se ova metoda prozvala izgleda ovako: *adresa_poslužioca/bugzilla/update?user=username&bugid=identifikator_greške*.

- **resolve()**: metoda za označavanje greške kao rešene; jedini neophodan parametar je identifikator greške dok opis greške nije neophodan, ali je moguć. URL na koji je potrebno uputiti HTTP POST zahtev da bi se ova metoda prozvala je:
adresa_poslužioca/bugzilla/resolve?user=username&bugid=identifikator_greške

4.2 Najbitnije klase klijentske aplikacije

Za realizaciju klijentske aplikacije korišćen je Argparse, Python-ov modul za pravljenje konzolnih aplikacija. On je deo standardne biblioteke tako da za njega ne postoji nikakvo dodatno podešavanje već je dovoljno samo uključiti modul. Pošto je i klijentska aplikacija u suštini proširiva odnosno treba da omogući jednostavno dodavanje novih komandi samim tim i novih klasa koje će biti zadužene za komunikaciju sa određenim servisima, i u klijentskoj aplikaciji je korišćen Yapsy sistem za dodatne module.

Struktura unutar klijentske aplikacije dosta podseća na strukturu klasa na samom poslužiocu. To je pre svega zato što se komande iz klijentske aplikacije preslikavaju direkno na metode poslužioca koje će se pozivati od strane klijenta. Drugim rečima, za svaku funkcionalnost nekog servisa postoji tačno jedna komanda sa klijentske strane.

4.2.1 Klasa Service_manager

Analogno istoimenoj klasi na serverskoj strani, klasa *Service_manager* je zadužena za rukovanje svim klasama koje komuniciraju sa određenim servisima. Kao što je za očekivati uključivanje novih servisa u sistem na serverskoj strani, tako je potrebno i u klijentskoj aplikaciji omogućiti da se jednostavno uključe novi moduli za komunikaciju sa njima.

Najvažnije metode su iste kao i u istoimenoj klasi na strani poslužioca:

- **get_service(name)**: metoda koja kao argument uzima ime servisa i ukoliko je servis registrovan vraća instancu klase tog sersvisa, u suprotnom, vraća vrednost *None*. Kada se vrednost *None* pojavi, znači da je došlo do neke greške u samom modulu i korisniku neće biti omućeno da kontaktira dati servis

- **get_service_list()**: metoda bez argumenata koja vraća listu svih registrovanih servisa

- **update_services()**: metoda bez argumenata koja osvežava podatke u listi svih registrovanih servisa

4.2.2 Klasa Service

Klasa Sevice predstavlja osnovnu i korensku klasu za modelovanje svih servisa od kojih klijentska aplikacija može zatražiti funkcionalnosti. U ovoj klasi se nalaze informacije koje su potrebne za komunikaciju sa poslužiocem poput adrese odnosno URL-a poslužioca.

Najbitnije metode ove klase su:

- **set_subparser(subparsers)**: metoda koja je u sprezi sa modulom Argparse unutar koje su definisana pravila za parsiranje komandi.

- **get_arguments(args)**: metoda koja preuzima argumente iz komandne linije i oblikuje ih u format pogodan za korišćenje dalje u aplikaciji

Svaki od servisa definiše svoja pravila spram parametara koji su potrebni, te redefiniše ove dve metode i dodaje nove za specifične funkcionalnosti.

4.2.3 Klasa Bugzilla

Klasa koja apstrakuje komunikaciju sa veb servisom Bugzilla. Nasleđuje klasu *Service* i dodaje sve metode neophodne za pokrivanje funkcionalnosti koje pruža servis. U okviru metode *set_subparser* se definišu sve ove funkcionalnosti, odnosno pobrojane su sve funkcije, prikazan

je opis kako se koriste, koji su parametri obavezni i neobavezni kao i pomoćne informacije za svaki parametar. Unutar metode *get_arguments* se preuzimaju svi unapred poznati argumenti (kao što je na primer korisničko ime) kao i oni koje korisnik treba da unese preko terminala. Takođe uz klasu ide i odgovarajući yapsy-plugin opis modula sa svim neophodnim podacima. Neke od metoda specifičnih za ovu klasu su:

- **list_bugs()**: metoda koja zahteva listu svih grešaka na kojima radi programer; nema dodatnih parametara. Zadatak ove metode je da iz radnog direktorijuma pokupi korisničko ime korisnika i uputi zahtev na URL *adresa_poslužioca/bugzilla/list?user=username*. Komandu koju je potrebno prozvati kako bi se ova metoda prozvala je *dpt bugzilla list*.

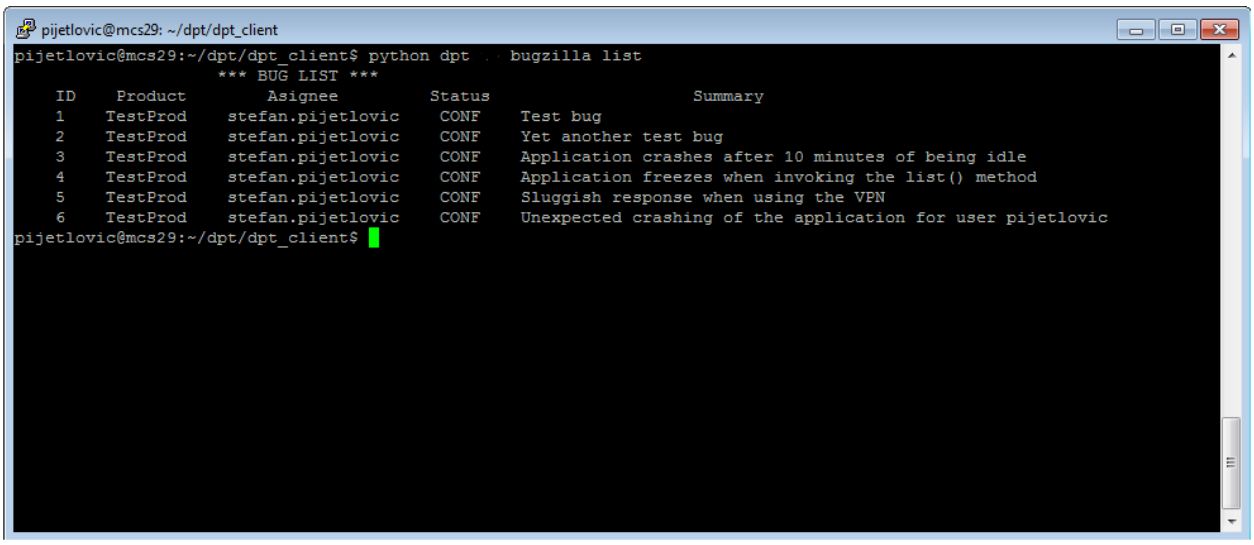
- **info()**: metoda koja zahteva identifikator greške; rezultat su detaljne informacije o toj grešci kao što su datum kada je greška prijavljena, detaljan opis greške, ključne reči vezane za grešku, prioritet odnosno ozbiljnsot greške (da li je greška trivialna ili u potpunosti ugrožava rad aplikacije), koliko je vremena neko proveo radeći na njenom ispravljanju kao i mnoge druge. Upućuje zahtev na *adresa_poslužioca/bugzilla/info?bugid=identifikator_greške* a odgovarajuća komanda je *dpt bugzilla info --bug bug_id*.

- **new()**: metoda koja pravi zahtev za evidentiranje nove greške; neophodni parametri su ime greške kao i opis greške koji se dostavlja kao tekstualni dokument. Upućuje zahtev na *adresa_poslužioca/bugzilla/new?user=username&bug=ime_greške*. Komandu koju je potrebno pozvati je *dpt bugzilla new --description putanja_do_datoteke*.

- **update()**: metoda koja pravi zahtev za ažuriranje stanja greške; neophodni parametri su identifikator greške kao i opis izmena koji se dostavlja u tekstualnom formatu kao i prilikom kreiranja zahteva za prijavu nove greške. Komanda koju je potrebno pozvati je *dpt bugzilla update --bug bug_id --description putanja_do_datoteke* a adresa kojoj se upućuje zahtev izgleda ovako:
adresa_poslužioca/bugzilla/update?user=username&bugid=identifikator_greške.

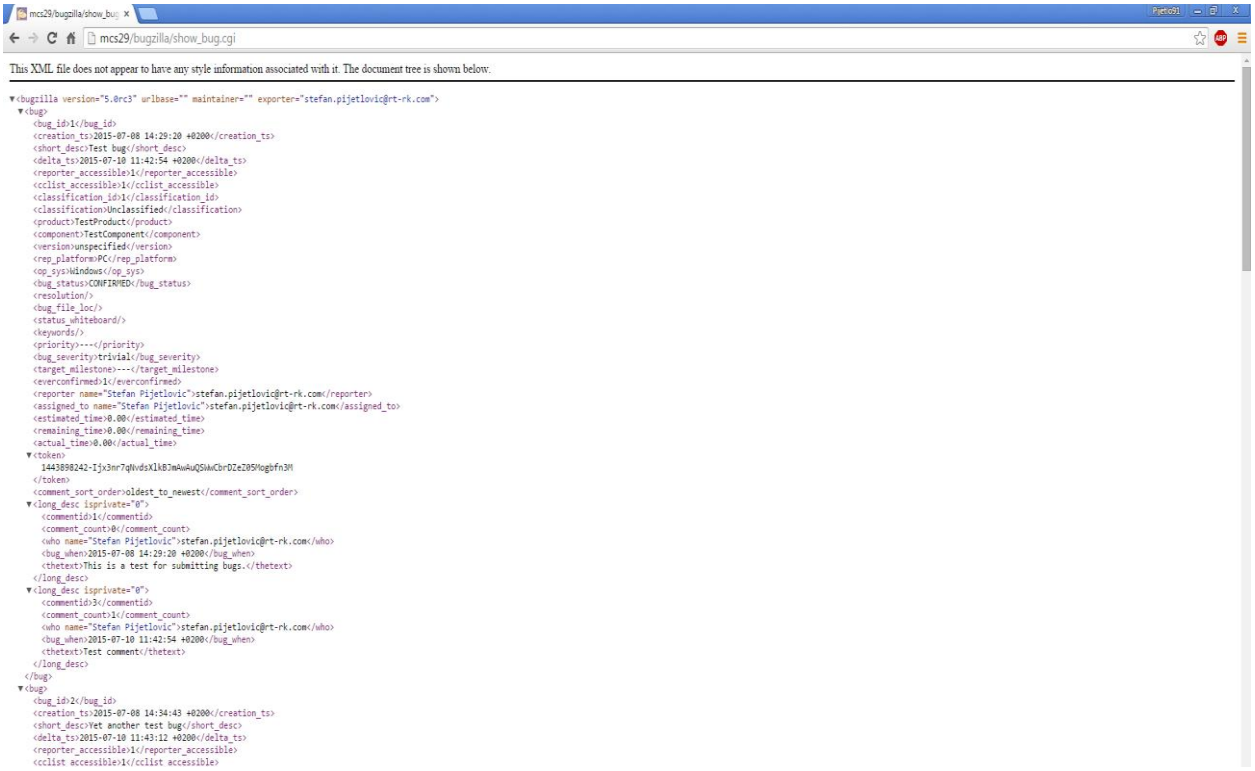
- **resolve()**: metoda koja pravi zahtev za rešavanje greške i njenog uklanjanja iz sistema; neophodni parametar je identifikator greške. Komanda glasi *dpt bugzilla resolve --bug bug_id* a URL je *adresa_poslužioca/bugzilla/resolve?user=username&bugid=identifikator_greške*. Opis je neobavezan ali je omogućen u slučaju da je potrebno dati neke specifične detalje.

Kao što je prikazano na slici 5 i objašnjeno u prethodnoj glavi, svaka komanda počinje ključnom reči *dpt* koja pokreće glavnu *Python* skriptu. Ova skripta kao argument uzima ime servisa. Svaki servis je takođe u suštini jedna *Python* skripta koja kao argument uzima imena funkcija kao i sve njihove neophodne parametre. Metodu servisa koju želimo ide odmah nakon imena servisa. Svi ostali parametri se navode uz pomoć "--" i željene vrednosti kao što je na primer *--bug_id 4*.



Slika 6 - Prikaz izlistavanja grešaka unutar aplikacije

Na slici vidimo primer izlistavanja grešaka. Da se primetiti da je aplikacija poprilično pregledna i nudi samo osnovne i najneophodnije informacije kao što su proizvod/projekat, zaduženi za grešku, status greške kao i njen naziv. Ukoliko korisnik želi više detalja o nekoj specifičnoj grešci, dovoljno je da pošalje zahtev na metodu za izlistavanje pojedinačne greške sa identifikatorom koji može očitati sa terminala.



Slika 7 - XML odgovor liste grešaka, znatno nepregledniji od opisa u aplikaciji

Iako je XML format izmišljen kako bi omogućio prikaz podataka koji su čitljivi i za ljude i za mašine, na slici 7 vidimo da taj prikaz ipak izgleda nečitko. Ukoliko se ovakva datoteta otvori u terminalu, nije jednostavno kretati se kroz nju i brzo pronaći neku informaciju, pogotovo ukoliko ne znamo tačno šta tražimo kao što je slučaj kada ne znamo tačan naziv greške koju tražimo ili njen identifikator. Alternativa XML-u je HTML format koji je znatno pregledniji ako se otvori u veb pretraživaču. Moguće je podesiti različite fontove, različite boje i ostale pogodnosti koje čine prikaz prijatnijim za oko. Međutim, ukoliko korisnik radi preko VPN-a, nekada mu nisu svi mrežni prolazi odobreni zbog bezbednosnih razloga, pa samim tim nekada nije moguće koristiti veb pretraživač. U takvoj situaciji ova klijentska aplikacija jeste najbolje rešenje.

5. Ispitivanje i evaluacija

Prilikom ispitivanja ovakvog sistema potrebno je uzeti u obzir da sistem ima mnogo komponenti, te da odziv sistema zavisi od više faktora kao što su:

- količina informacija u bazi podataka svakog od servisa
- broja korisnika koji koriste sistem
- gustine internet saobraćaja
- mogućnosti odnosno performanse fizičke arhitekture računara na kojima se nazale servisi, poslužilac pa i klijentske aplikacije.

Svi ovi faktori utiču u manjoj ili većoj meri na brzinu odziva s obzirom na potencijalno dugačak put koje informacije ponekad moraju preći. Treba napomenuti da je sistem i dalje u fazi razvoja što uvodi neka ograničenja. Iz ovih razloga, ispitivanje smo podelili u dva slučaja.

U prvom je korišćen idealizovan scenario u strogo kontrolisanim uslovnima, odnosno mali broj korisnika sa relativno malo podataka u bazi podataka gde se i poslužilac i klijent nalaze u lokalnoj mreži i mereno je vreme odziva sistema. Ovaj slučaj nije realan jer ovakvi servisi često imaju na stotine, a neretko i hiljade korisnika, dok se vremenom u bazi podataka nađe izuzetno velika količina informacija. Isto tako, ukoliko neka kompanija ima neznatan broj korisnika sa relativno malom količinom podataka, rešenje predstavljeno ovim radom verovatno nije ni potrebno. Ipak, može da služi kao referentna tačka da prikaže koliko sistem zaostaje za nekim idealnim vrednostima.

Drugi slučaj predstavlja realniji scenario odnosno scenario kakav je potreban da bi ovakva arhitektura imala svoju pravu vrednost. Komunikacija se ovoga puta odvija preko VPN-a, klijent se nalazi u Srbiji a poslužilac u Sjedinjenim Američkim Državama. U ovom slučaju je i obim podataka znatno veći jer u bazama podataka servisa ima nekoliko hiljada korisnika sa različitim brojem prijaveljnih grešaka u rasponu od jedne do nekoliko desetina.

Ni u jednom ni u drugom testnom slučaju nisu bili uključeni raspoređivač zahteva niti keširanje kako bi se saobraćaj sveo na minimum. Klijentska aplikacija je testirana na laptop računaru Fujitsu Lifebook AH522/SL sa sledećim performansama:

- procesor Intel i7-3612QM sa 4 jezgra (8 niti) i radnim taktom od 2.1 GHz
- 6 GB radne memorije
- operativni sistem na računaru je Windows 7

Performanse računara na kome je instaliran naš poslužilac su sledeće:

- 8 procesora tipa Intel i7-4770K, svaki sa po 4 jezgra (8 niti) i radnim taktom od 3.5 GHz
- 16 GB radne memorije
- operativni sistem Linux Ubuntu 14.04

5.1 Idealizovan slučaj

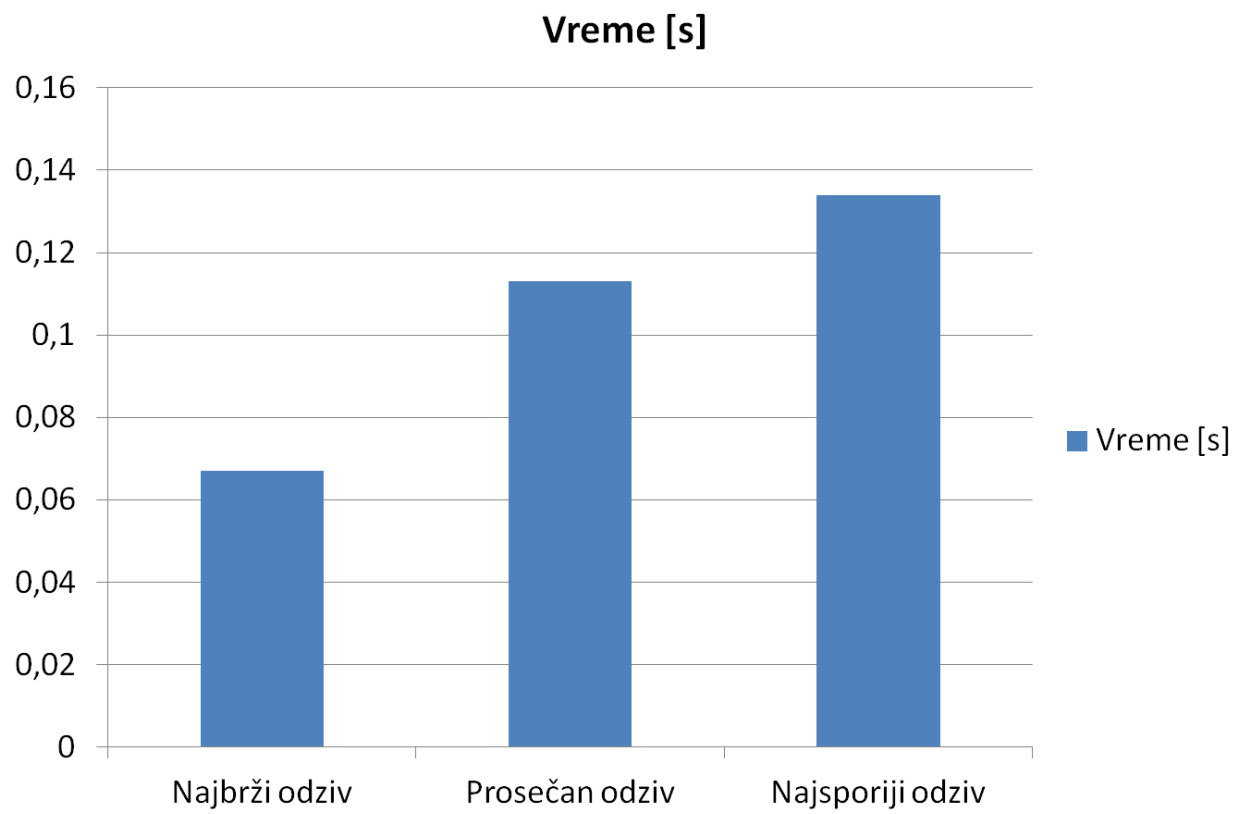
Prilikom ispitivanja ovog slučaja je napravljeno nekoliko desetina merenja vremena odziva servisa za praćenje grešaka. Svaki put je ispitivana metoda za izlistavanje grešaka i uvek za istog korisnika, kako bi količina podataka uvek bila ista i ne bi uticala na različitu količinu informacija koje je na kraju potrebno poslati, isparsirati i ispisati na ekran.

Rezultati su sumirani u tabeli ispod u tri kolone i to kao prosečna vrednost, najbrže i najkraće vreme odziva izraženo u sekundama.

Veličina	Vreme odziva u sekundama
Najbrži odziv	0.067
Prosečan odziv	0.113
Najsporiji odziv	0.134

Tabela 1 - Merenja u idealizovanom slučaju

U rezultatima se vidi da postoje odstupanja ali da su ona mala i da korisnik nije u stanju da ih primeti zbog tromostih ljudskih čula. S obzirom da se radi u kontrolisanim uslovima i da praktično ne postoji kašnjenje unutar mreže, razlika se najverovatnije pojavljuje usled raspoređivanja procesa odnosno niti unutar procesora poslužioca. Razlog za ovakvu pretpostavku je taj što je veliki broj slučajeva bio ili u vremenskom rasponu između 0.067 i 0.07 ili 0.131 i 0.134 sekundi. U realnom sistemu nije za očekivati ovako male varijacije između najbržeg, prosečnog i najsporijeg odziva, što se vidi u narednom poglavlju.



Slika 8 - Grafički prikaz vremena odziva u idealizovanom slučaju

5.2 Realan slučaj

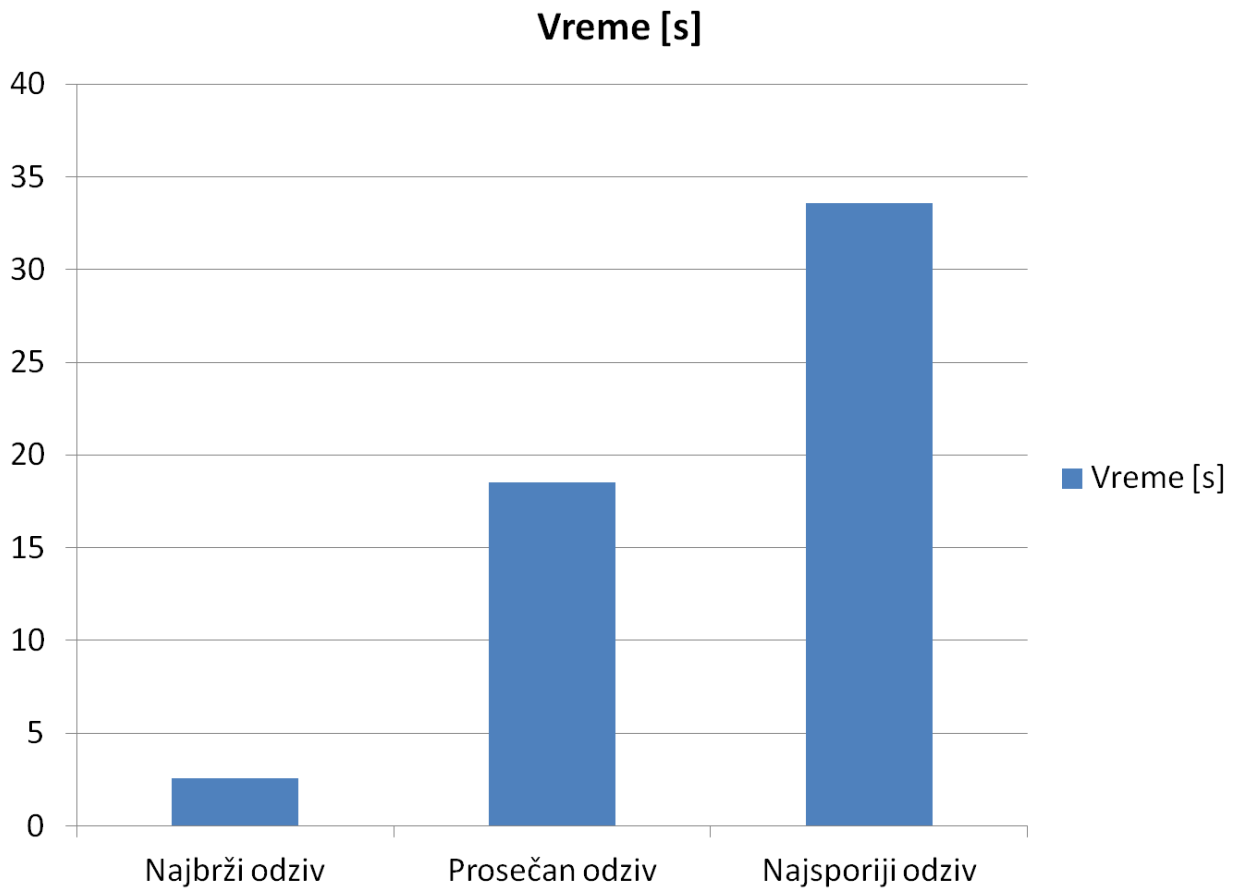
Prilikom ispitivanja ovog slučaja je takođe napravljeno nekoliko desetina uzastopnih merenja vremena odziva servisa za praćenje grešaka. Isto kao i u prethodnom slučaju je ispitivana metoda za izlistavanje grešaka za istog korisnika, kako ne bi imali promenljivu količinu informacija koju je potrebno poslati preko mreže i prikazati na ekranu. Prilikom ovog slučaja treba uzeti u obzir da postoje faktori koji utiču na znatno lošije rezulate u odnosu na idealizovan slučaj i da na neke od njih kao što su gustina saobraćaja i preusmeravanje paketa zbog VPN-a ne možemo uticati poboljšanjem klijentske aplikacije i poslužioca.

Rezultati merenja su prikazani u tabeli ispod analogno prethodnom merenju.

Veličina	Vreme odziva u ms
Najbrži odziv	2.592
Prosečan odziv	18.529
Najsporiji odziv	33.582

Tabela 2 - Merenja u realnom slučaju

Prva stvar koja se može primetiti u odnosu na rezultate iz prethodnog slučaja jeste izuzetno velika razlika između najbržeg i najsporijeg zabeleženog odziva sistema. Isto tako i varijacije u odnosu na prosečno vreme odziva su veoma izražene. S obzirom da je količina informacija koju je potrebno proslediti mrežom i ispisati na ekran identična, jasno je da to nije uzrok ovoliko lošijeg odziva. Kada pridodamo i činjenicu da je ispitivanje obavljeno uz pomoć računara istih performansi, razloge za ovoliko slabije rezultate treba potražiti u količini informacija u bazama podataka koje je potrebno pretražiti i u kašnjenju zbog prenosa informacija na veliku udaljenost.



Slika 9 - Grafički prikaz vremena odziva u realnom slučaju

Ispitivanje je obavljeno u relativno kratkom vremenskom periodu tako da se količina informacija u bazi podataka nije menjala ili se promenila zanemarljivo malo, reda veličine nekoliko novih prijavljenih grešaka spram nekoliko desetina hiljada već registrovanih. Samim tim, možemo smatrati da je skup koji je potrebno pretražiti ostao nepromenjen. Ako aplikacija uvek pretražuje isti skup nije realno za očekivati da će razlike između dve uzastopne pretrage biti preko 30 sekundi.

Uzimajući u obzir sve do sada navedeno, nameće se zaključak da je za lošije rezultate najviše odgovoran prenos informacija na veliku udaljenost kroz veliki broj internet čvorova. Svi

ostali faktori koji utiču na odziv su u najvećoj meri konstantni, tako da saobraćaj u internetu ostaje jedina promenljiva na koju, nažalost, ne možemo uticati.

Iako rezultati deluju na prvi pogled obeshrabrujuće spram idealnih rezultata, treba razmotriti i način na koji će se sistem koristiti. Držaćemo se prosečnih vrednosti zarad opšteg slučaja korišćenja. Naime, jednom kada korisnik u konzoli ukuca komandu za izlistavanje grešaka na kojima radi, dobiće rezultat u proseku u roku od 18.5 sekundi. Dokle god ne ugasi aplikaciju, njemu su te informacije dostupne i neće se menjati sve dok ih on sam ne promeni. Samim tim, korisnik neće imati potrebu da veliki broj puta u toku dana izvršava ovu komandu. Isto tako, nije za očekivati da će korisnik rešiti ili ažurirati veliki broj grešaka u toku jednog radnog dana. Tako da je vreme utrošeno za dobavljanje svih potrebnih informacija za rad prihvatljivo.

6. Zaključak

U radu je predstavljen jedan pristup objedinjavanju alata za razvoj programske podrške napisan u programskom jeziku Python. Uz pomoć ovakog sistema moguće je na uniforman i jednostavan način efikasno rukovati raznorodnim alatima. Takođe je skraćeno vreme obuke za programere koji se nađu u novom radnom okruženju bilo da je u pitanju stalno zaposlednje ili naprosto najamni rad za neku drugu kompaniju. Povećanje produktivnosti se postiže kroz apstrakciju svih servisa kroz jednostavnu i preglednu konzolnu aplikaciju i činjenicom da programeri sve aktivnosti neophodne za razvoj mogu sprovesti bez napuštanja terminala.

Upotrebom Flask mikro okruženja za razvoj veb aplikacija zajedno sa Yapsy sistemom za dodatne module je omogućeno jednostavno i lako dodavanje novih servisa u sistem kao i zamena postojećih jer su servisi realizovani kao dodatni programi. Korisnici ne moraju biti svesni da je do ovih promena došlo. Svi servisi prate identičnu strukturu pošto su potomci zajedničke klase. Isto tako je omogućena i realizacija više vrsta klijentskih aplikacija.

Zadatak rada je uspešno ispunjen i time su otvorene nove mogućnosti za dalja unapređenja i proširenja sistema, pre svega sa klijentske strane.

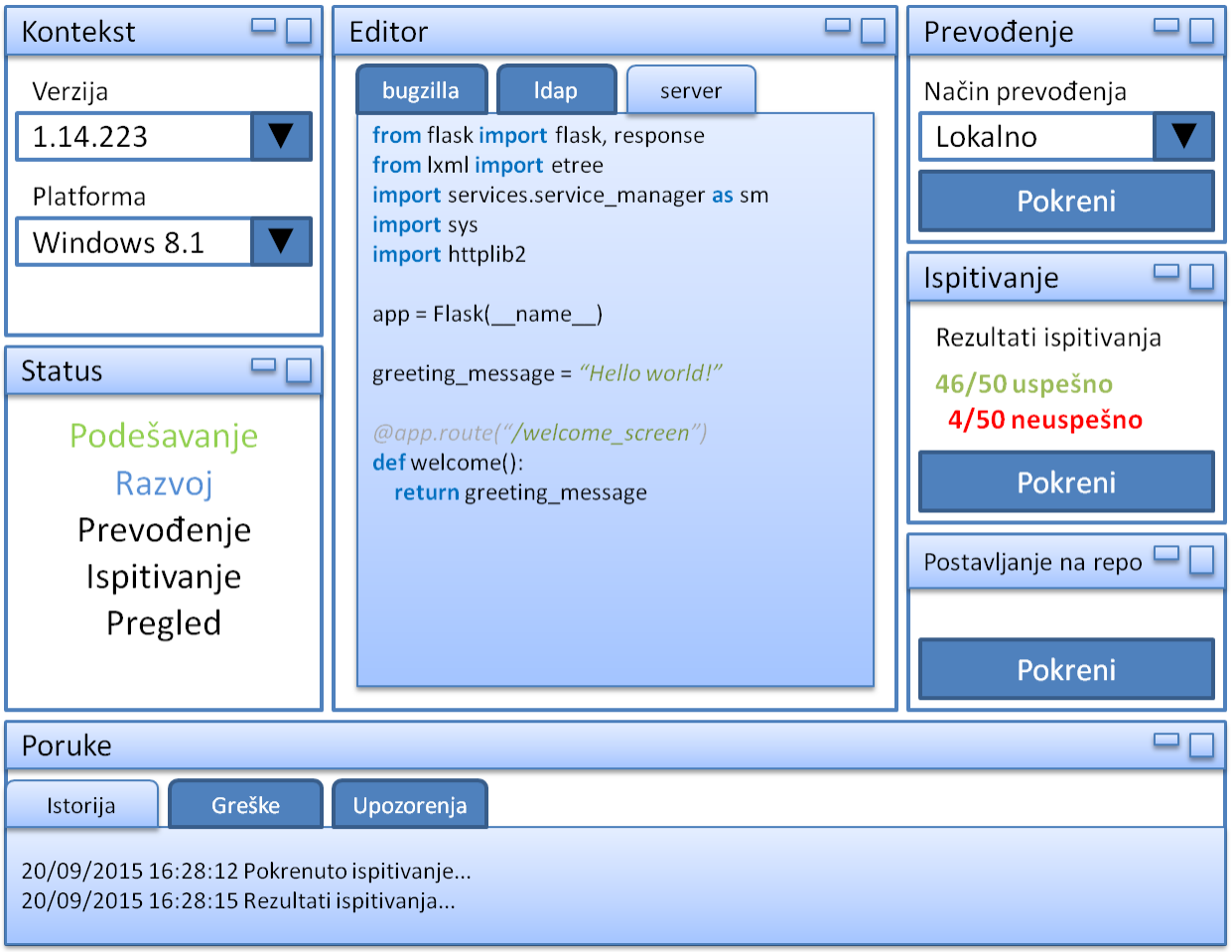
6.1 Mogući dalji tokovi razvoja

6.1.1 Integracija sa već postojećim razvojnim okruženjima

Jedno od mogućih rešenja jeste ugradnja aplikacije u neko već postojeće razvojno okruženje kao što je Eclipse. Eclipse sadrži osnovni radni prostor koji se sastoji od naprednog tekstualnog editora, skupa alata koji su potrebni za razvoj programske podrške kao što su

prevodilac, povezivač, dibager i proširivim sistemom za dodatne programe. Napisan je najvećim delom u programskom jeziku Java i koristi se najčešće za razvoj Java aplikacija, ali se mogu koristiti i drugi programski jezici, između ostalih i Python [9].

Celu klijentsku aplikaciju je moguće realizovati kao jedan dodatni program kao što je prikazano na slici.



Slika 10 – Izgled aplikacije realizovane kao dodatni program za Eclipse

Napredni editor Eclipse-a i dalje zauzima centralnu figuru okruženja ali su ostali prozori zamenjeni. U okviru ove perspektive, programer odmah prilikom otvaranja ima pregled koraka tokom razvoja. Moguće je izabrati željenu verziju aplikacije, izabrati opcije za prevođenje da li da bude lokalna ili na nekom poslužiocu, pokrenuti sve testove kako bi se utvrdila ispravnost koda. Korak dalje bi bio onemogućiti postavljanje koda na repozitorijum ukoliko neki određeni testovi nisu prošli.

Korisnik može da manipuliše sa svim prozorima kao i do sada tako što menja njihove dimenzije i raspored, može ih skloniti ukoliko mu smetaju i zameniti ih nekim drugim. Na ovakav način, programeri koji su već upoznati sa radom u ovom veoma poznatom okruženju otvorenog koda ostvaruju uštedu u vremenu jer ne moraju napuštati okruženje kako bi se bavili svim ostalim aktivnostima koje prate razvoj.

Prednosti ovakve realizacije su višestruke. S obzirom da se radi o već poznatom okruženju, period privikavanja za rad i korišćenje aplikacije je znatno kraći nego kada je slučaj sa potpuno novom aplikacijom. Takođe postoji i velika zajednica koja razvija dodatne module za Eclipse pa samim tim i velika količina lako dostupnog znanja. Sam Eclipse pruža mnogo mogućnosti za razvoj komponenti neophodnih za ispravno funkcionisanje aplikacije.

S druge strane, postoje i određene mane prilikom ovakvog pristupa. Pre svega, očigledna je velika zavisnost od samog Eclipse-a. U slučaju da ovaj projekat prestane da postoji u nekom trenutku, blokiran je i potencijalni dalji razvoj aplikacije. Eclipse ima i neke svoje zahteve koji nisu neophodni za rad naše aplikacije kao što je instalacija Jave na računar i nezanemarljivo zauzeće kako radne, tako i masovne memorije koje je znatno veće nego što je potrebno za funkcionalnosti naše aplikacije.

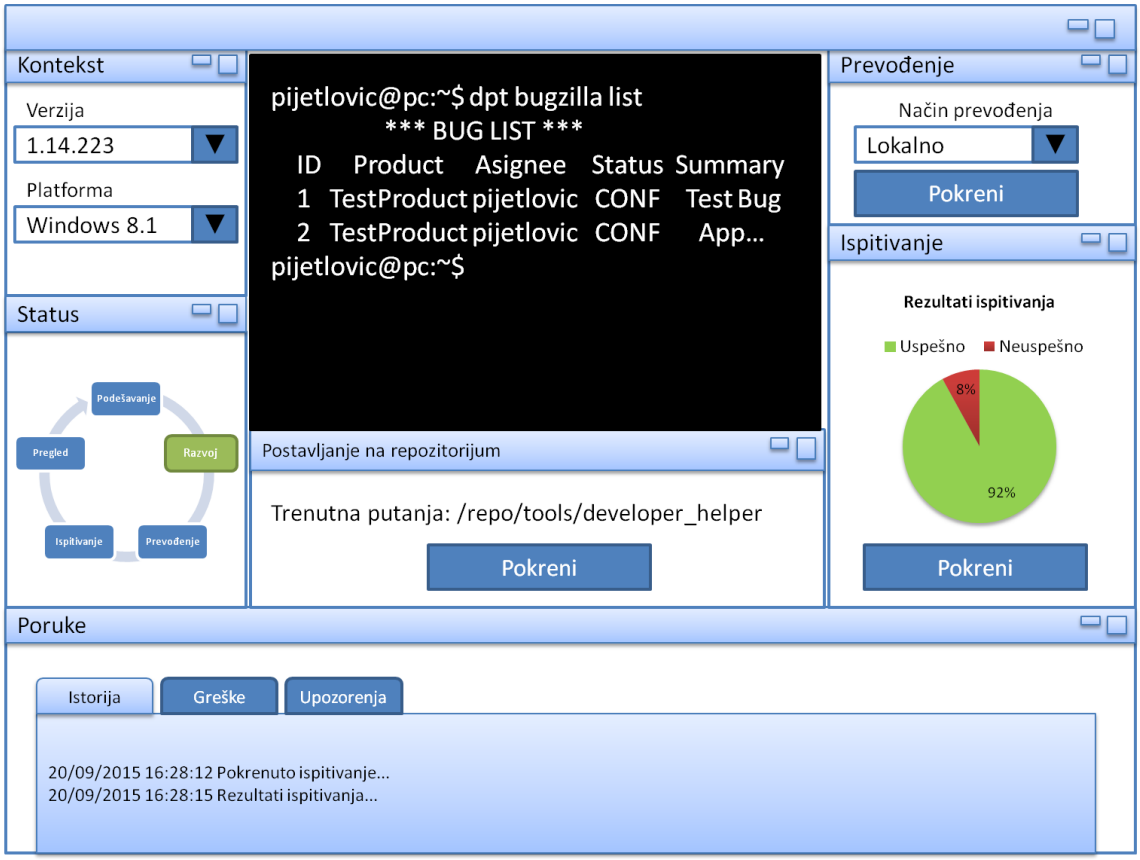
6.1.2 Samostalna grafička ili aplikacija u veb pretraživaču

Ukoliko je prevelik problem zavisnost od nekih drugih orkuženja, alternativa je napraviti svoju nezavisnu grafičku ili veb aplikaciju. Podešavanja trebaju biti identična kao i kod konzolne aplikacije samo umesto kucanja komandi korisnici se mogu kretati kroz ceo proces razvoja podrške uz pomoć miša i tastature. Razvoj sopstvene aplikacije omogućuje i veću slobodu kada su u pitanju grafičke komponente koje se mogu nalaziti unutar aplikacije.

I jedan i drugi pristup imaju svoje prednosti i svoje mane. Ukoliko klijentsku aplikaciju realizujemo kao veb aplikaciju, velika prednost je to što na računarima na kojima će raditi korisnici nije potrebno instalirati ništa više osim veb pretraživača koji obično postoje na svakom računaru. Još jedna velika pogodnost je ta što u slučaju potrebe za ažuriranjem aplikacije (pojava novih servisa u sistemu ili izmena nekih postojećih) se sve obavlja samo na poslužiocu koji je domaćin računar za aplikaciju. Korisnik ne mora ništa da ažurira. Što se samog razvoja tiče, jednostavnije je podržati da veb aplikacija radi podjednako dobro na najpoznatijim veb čitačima nego razvijati aplikaciju tako da radi podjednako dobro na svim poznatim operativnim sistemima i svim njihovim aktivnim verzijama.

Jedna od osnovnih mana pri ovakvoj realizaciji je pre svega brzina. Koliko god veb pretraživači bili napredni i brzi, oni su ipak samo procesi koji se izvršavaju na nekom operativnom sistemu, tako da nikada ne mogu u potpunosti parirati samostalnim aplikacijama po brzini. Takođe, jedna od otežavajućih okolnosti jeste i obavezno prisustvo interneta na računaru na kome radi korisnik. U slučaju da dođe do prekida, nije moguće kontaktirati poslužioc a ne

postoji način da iskoristimo neke starije informacije o stanju ukoliko korisnik nije prethodno sačuvao sadržaj stranice.



Slika 11 - Izgled samostalne aplikacije

Prednosti i mane realizacije samostalne aplikacije su u suštini inverzija prednosti i mana kod veb aplikacije. Prednosti su pre svega brzina koja je veća kao i mogućnost da se jednostavno sačuva poslednje stanje aplikacije jednom kada se ona ugasi bez da korisnik o tome vodi računa. U slučaju prekida na mreži, korisnik tada i dalje može da radi sa kopijom starih informacija što je nekada dovoljno. Postavlja se pitanje koliko su ove prednosti zaista velike s obzirom na relativno malu količinu podataka koju je potrebno obraditi sa klijentske strane. Zapravo, klijentska aplikacija obezbeđuje samo prikaz i komunikaciju sa servisima. Tako da ni mogućnosti koje samostalna aplikacija nudi spram veb aplikacija u vidu grafike nisu bitne jer je primarna namena aplikacije da omogući jednostavan, brz i intuitivan razvoj programske podrške. Nisu potrebni nikakvi grafički napredni elementi koji bi mogli samo da odvlače pažnju.

Mane u odnosu na veb aplikacije nisu nezanemarljive. Pre svega, jednostavnije je voditi računa o tome da aplikacija radi na različitim veb čitačima nego na različitim operativnim sistemima s obzirom na to da je mnogo lakše da korisnik kod sebe instalira neki veb čitač koji je podržan nego da nabavi novi operativni sistem. Druga mana se vidi kod ažuriranja. Naime ukoliko sistem koristi nekoliko hiljada ili desetina hiljada korisnika, potrebno je da svaki od njih pojedinačno ažurira aplikaciju na svom računaru. Čak i u slučaju da je ažuriranje automatsko,

potrebno je da protekne određeno vreme za to. Kada se to vreme pomnoži sa nekoliko hiljada, dobija se značajan period praznog hoda.

6.1.3 Proširenja na strani poslužioca

Moguća proširenja na strani poslužioca su mnogobrojna. Pored dodavanja velikog broja novih servisa, unapređenja je moguće napraviti na raspoređivaču zahteva, u internoj bazi podataka za skladištenje zahteva i mnoge druge. U slučaju izuzetno velikog broja korisnika, poslužilac se može rasporediti na više mašina, tako da je moguće paralelno uslužiti veći broj korisnika.

Još jenda od mogućnosti je uvođenje nekog vida statistike. Pratilo bi se ponašanje korisnika u vidu koji se servisi najčešće koriste, koje funkcionalnosti, koji je stepen otkaza ili neuspelih zahteva i tako dalje. Sve ove informacije se mogu dalje koristiti za poboljšanje pojedinačnih servisa.

7. Literatura

[1] TIOBE Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, učitano 15.10.2015.

[2] PEP 20 --The Zen of Python, https://www.python.org/dev/peps/pep-0020, učitano 15.10.2015.

[3] Flask documentation, http://flask.pocoo.org/docs/0.10, učitano 15.10.2015.

[4] Argparse documentation, https://docs.python.org/3/library/argparse.html, učitano 15.10.2015.

[5] Yapsy documentation, http://yapsy.sourceforge.net, učitano 15.10.2015.

[6] Fielding, Roy T.; Taylor, Richard N. (May 2002), "Principled Design of the Modern Web Architecture", ACM Transsactionss on Internet Technology (TOIT) (New York: Association for Computing Machinery) 2(2): 115-150

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal, "Pattern Oriented Software Architecture Volume 1: A System of Patterns", 1996.

[8] Bugzilla documentation, https://bugzilla.readthedocs.org/en/5.0/, učitano 15.10.2015.

[9] Eclipse Luna documentation, http://help.eclipse.org/luna/index.jsp, učitano 15.10.2015.