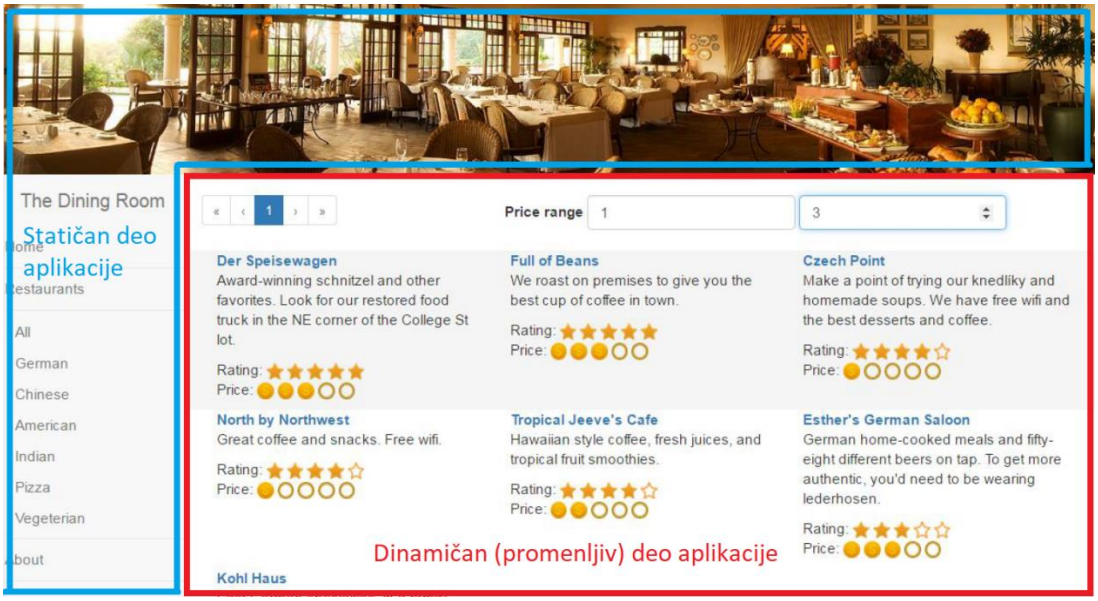
Dekompozicija teksta zadatka

Prvi korak kod formiranja front-end veb-aplikacije jeste da se temeljno pročita specifikacija zahteva i sagleda predložen izgled interfejsa, kako bi se **uočile celine aplikacije i formirao spisak komponenti sa kojim ćemo raditi**. Nije neophodno uočiti u startu sve komponente. Za početak, treba razmotriti koje komponente predstavljaju srž aplikacije, odnosno koje komponente ćemo generisati za **core paket**.

> **Core paket**
>
> Core paket sadrži komponente koje predstavljaju statički deo veb-aplikacije, odnosno komponente koje su uvek prisutne bez obzira u kom delu aplikacije se nalazimo. Primeri za ovo su **header**, **footer**, i **navbar** komponente. Pored toga, core paket može da sadrži i komponente koje nisu vezane za neki konkretan entitet već čine srž naše veb-aplikacije. Primer za ovo bi bila **homepage**, **about** i **contact** komponenta. Najzad, core paket može biti dobro mesto da se grupišu servisi koji se koriste kroz celu aplikaciju, poput servisa za logovanje i obradu grešaka (**logger**). *Angular style guide* daje dobre savete za rad sa core paketom: https://angular.io/guide/styleguide#core-feature-module.

Čitajući tekst zadatka, vidimo da je potrebno izraditi aplikaciju koja ima tri strane. Za svaku stranu računamo da ćemo imati jednu ili više komponenti koje treba napraviti, a koliko tačno otkrivamo tako što analiziramo konkretnu stranicu. Pored toga, znamo da ćemo imati nekoliko komponenti za statički deo aplikacije. Razmatranjem prve slike iz teksta zadatka možemo da uočimo šta je statički deo aplikacije, a šta dinamički i ova podela je prikazana u slici 1.



Slika 1 Identifikacija statičnih komponenti i dinamičnog sadržaja

Statički deo *Dining Room* aplikacije možemo podeliti na dve celine, sliku koja se nalazi u zaglavlju, koja prikazuje restoran, i meni koji se nalazi sa strane. Možemo da zamislimo da su ovo **HeaderComponent** i **SidebarComponent** komponente.

Dinamički deo aplikacije prikazuje tri različite stranice. Svaka stranica će biti predstavljena jednom komponentom, te možemo zamisliti **HomeComponent**, **AboutComponent** i **RestaurantComponent**. Iz teksta zadatka vidimo da su Home i About komponente trivijalne i da sadrže samo tekst, dok će RestaurantComponent sadržati više podkomponenti i pružati složenije ponašanje.

Pre nego što krenemo u detaljniju analizu Restaurants komponente, možemo primetiti da smo definisali većinu (ako ne i sve) komponente koje će nam trebati za core paket, i to su:

- HeaderComponent i SidebarComponent – statički deo sajta;
- HomeComponent i AboutComponent – jednostavne stranice koje čine srž aplikacije, i ne odnose se na neki konkretan entitet, poput restorana.

Činjenica da imamo različite stranice da prikazujemo u zavisnosti od putanje nam je znak da nam je potrebno rutiranje u aplikaciji. Velika većina front-end aplikacija, ako ne i sve koje budete razvijali, će zahtevati makar bazično rutiranje. Za ovu aplikaciju možemo zamisliti da će postojati putanja **/home**, **/about** i **/restaurants**.

Često testiranje koda i rešavanje grešaka

Prilikom razrade aplikacije, cilj je da **pravimo što sitnije izmene i da što češće testiramo ispravnost tih izmena**. Česta greška koja se pravi kod učenja nove tehnologije jeste da se previše koda iskuca bez provere njegove ispravnosti. Ukoliko bi generisali aplikaciju, formirali sve komponente, uvezali rutiranje i tek onda testirali prvi put sistem, postoji dobra šansa da bi se pojavila greška koju bi teško otklonili, te bi neretko bili naterani da krenemo ispočetka postepeno. Kada se razvije poznavanje tehnologije i celokupna veština razvoja front-end aplikacija u redu je uzimati prečice. Međutim, u početku treba biti strpljiv i postepeno sve raditi u što sitnijim koracima.

Ako pravimo aplikaciju u sitnim koracima i dođemo u situaciju da nam se pojavi greška na ekranu ili u konzoli, u opštem slučaju ćemo moći da zaključimo da je poslednji mali korak koji smo napravili, odnosno poslednja komponenta ili deo koda koji smo napisali, glavni krivac. Svaki put kada se pojavi greška u konzoli potrebno je pročitati jednom ili više puta tekst greške, sa fokusom na prvih nekoliko redova, sa ciljem razumevanja gde je problem nastao. Ukoliko nije jasno šta bi mogao biti uzrok greške, *Google* može da pomogne, unošenjem ključnih reči iz teksta greške kao upit.

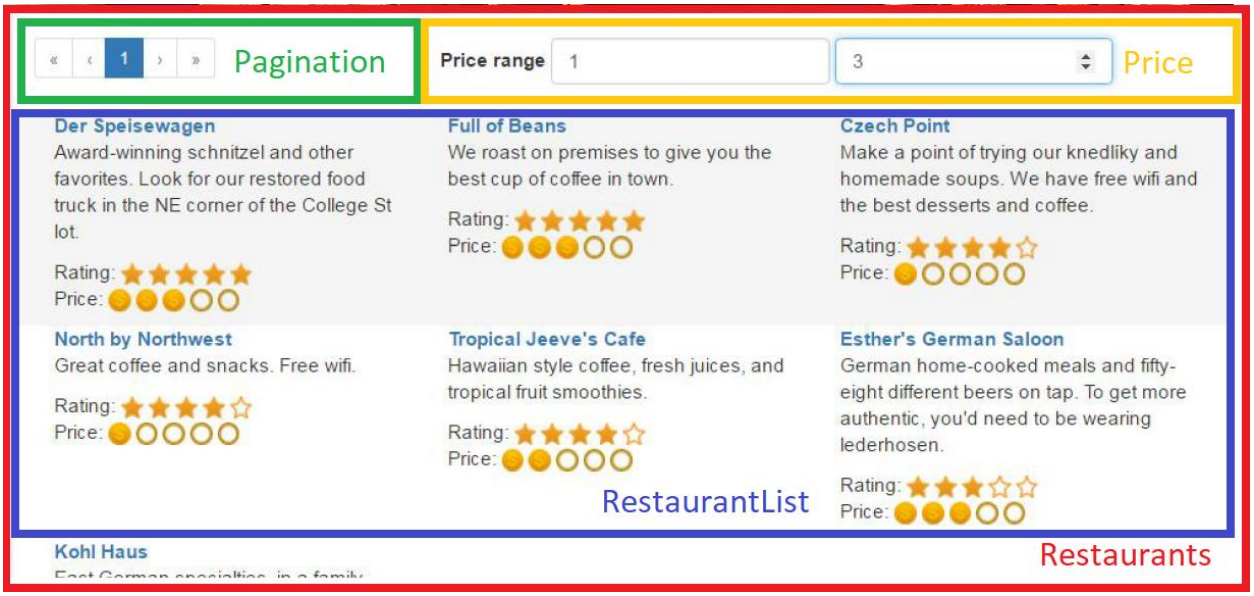
U ovom momentu možemo da krenemo da implemenitramo naše rešenje. Da bi implementirali sve što smo do sada naveli, potrebno je uraditi sledeće korake:

1. Generisanje nove *dining-room* Angular aplikacije upotrebom *angular-cli* alata;
2. Kreiranje core paketa sa osnovnim komponentama koje smo identifikovali;
3. Pravljenje elementarnog rutiranja kako bi prikazali sve komponente iz core paketa.

Ova tri početna koraka će biti u velikoj većini slučajeva isti za svaki novi projekat koji počinjete.

Dekompozicija komponente za prikaz restorana

Sada kada je postavljena infrastruktura, odnosno definisan core paket, analiziramo dalje aplikaciju koju treba da realizujemo. Posmatramo sliku 1 iz teksta zadatka i razmatramo spisak funkcionalnosti koji su taksativno navedeni ispod slike. Za početak možemo zamisliti da će cela stranica biti predstavljena nekom Restaurants komponentom, koja će se sastojati od nekoliko podkomponenti. Daljim razmatranjem uočavamo nekoliko celina aplikacije, koje možemo mapirati na komponente. Ovo je prikazano u slici 2.



Slika 2 Identifikovane podkomponente Restaurants komponente

Dakle, za sada su uočene tri veće celine koje čine **RestaurantsComponent**, i to su:

- **PaginationComponent** – komponentu koja omogućava straničenje podataka, te dobavlja sa servera parcijalne podatke kako bi izbegli učitavanje celokupnog sadržaja baze;
- **PriceComponent** – komponenta koja omogućava pretragu po ceni restorana, kako bi se filtrirala lista restorana;
- **RestaurantListComponent** – komponenta za prikaz liste restorana, gde je svaka stavka liste jedan restora.

Komponenta za straničenje i pretragu po ceni se sastoje od par HTML elemenata, te možemo da zaključimo da ove komponente neće imati dodatne podkomponente. Međutim, komponentu za prikaz liste restorana, zbog relativno kompleksnog sadržaja svake stavke, možemo zamisliti kao listu **RestaurantItemComponent** elemenata.

Najzad, zbog eksplicitnog navođenja u tekstu zadatka, vidimo da elementi koji prikazuju zvezdice za ocenu restorana i novčiće za njegovu cenu moraju biti komponente, gde su oba elementa predstavljena sa istom komponentom koja prihvata različit ulaz (npr. **RatingComponent**). Slika 3 ilustruje sve komponente koje smo identifikovali u okviru RestaurantList komponente.

Slika 3 Razlaganje RestaurantList komponente na podkomponente

Pred nama stoji ozbiljan zadatak. Za sada smo identifikovali čak šest komponenti koje trebaju međusobno da interaguju. To su:

- RestaurantsComponent;
- PaginationComponent;
- PriceComponent;
- RestaurantListComponent;
- RestaurantItemComponent;
- RatingComponent.

Pored toga, potrebno je dobaviti same podatke sa servera i omogućiti logiku da se novi zahtevi formiraju svaki put kada korisnik promeni stranicu ili izmeni cenu. Za ovo bi nam bio zgodan servis koji možemo zvati **RestaurantService**. Postavlja se pitanje u kom redosledu krenuti sa formiranjem identifikovanih komponenti i servisa. Kako pristupiti daljem rešavanju zadatka, tako da se držimo saveta da **što sitnije izmene pravimo, čiju ispravnost možemo što češće da testiramo.** Da bi ovo odredili moramo da još malo analiziramo komponente.

Za početak, potrebno je identifikovati koje komponente moramo da imamo da bi prikazali druge komponente, odnosno koje su roditeljske komponente. Vidimo da RestaurantsComponent sadrži preostalih pet komponenti, te ćemo nju morati prvu da definišemo. Takođe znamo da nema mnogo smisla definisati RestaurantItem komponentu, dok ne definišemo RestaurantList komponentu.

Međutim, šta se dešava kada imamo sestrinske komponente? Kako da znamo da li prvo da kreiramo RestaurantList, Pagination ili Price komponentu? Za ovo moramo detaljnije da razmotrimo šta svaka od ovih komponenti treba da radi i kako aplikacija i korisnik interaguju sa njima. Najbitnije, treba da shvatimo šta nam je potrebno da bi znali da komponenta ispravno radi.

Svrha RestaurantList komponente jeste da iterira kroz listu restorana (koja je dobijena preko servisa ili kao Input od roditeljske komponente) i da prikaže svaki restoran. Dakle, znamo da RestaurantList komponenta radi ako vidimo prikaz restorana koje smo joj prosledili. Ovaj prikaz možemo da vidimo bez paginacije i filtriranja po ceni, zbog čega zaključujemo da je ovo dobar kandidat za prvu komponentu koju ćemo implementirati, nakon Restaurants komponente.

Međutim, RestaurantList komponenta je zamišljenja da prikaže listu RestaurantItem komponente. Postavlja se pitanje da li moramo obe komponente prvo generisati, pa tek onda kada iskucamo HTML za obe i prosledimo odgovarajuće podatke svakoj testirati rešenje prvi put? Ovo zavisi od pojedinca. Ukoliko se držimo prakse pravljenja što sitnijih koraka u razvoju, prvi korak bi bio da se implementira RestaurantList komponenta koja će proći kroz listu restorana i prikazati svaki. Naredni korak bi mogao da bude da se formira RestaurantItem komponenta, pa da se izmeni RestaurantList tako da prosleđuje pojedinačni restoran svakoj instanci RestaurantItem komponente.

Iako imamo dobrog kandidata za prvi naredni korak (generiasanje i implementiranje Restaurants, pa zatim RestaurantList komponente), razmotrićemo još Price i Pagination komponentu. Da li bi neka od ove dve komponente bile pogodan kandidat za početak naše implementacije?

Price komponenta sadrži dva input HTML elementa, gde je ideja da korisnik unosi neki opseg cena i da se, u zavisnosti od njegovog unosa, prikažu restorani u RestaurantList komponenti čija cena upada u taj opseg. Da bi mogli da jednostavno testiramo ispravnost ove komponente, nama je potrebno da imamo prikaz restorana kako bi mogli da vidimo da se stvarno prikazuju oni čija cena spada u naveden opseg. Prema tome, potreban nam je RestaurantList.

Što se Pagination komponente tiče, korisnik klikom na taster menja stranicu sa idejom da se na prikazu restorana pojave oni koji pripadaju stranici koju je odabrao. Ovde opet imamo zavisnost Pagination komponente od RestaurantList komponente.

Napomena: Pagination komponenta se često pronalazi, kako u našim zadacima tako i u veb-aplikacijama koje pronalazite na internetu. Kako je *copy-paste* jedna od najčešćih tehnika kojim se služe programeri, možete skratiti sebi posao tako što ćete preuzeti kod ove komponente iz druge aplikacije. Međutim, vodite računa da nije dovoljno kopirati komponentu i pružiti osnovnu funkciju straničenja, već morate da prilagodite komponentu zahtevu zadatka. Neke komponente za straničenje će imati samo tastere Next i Previous, dok će druge imati First, Last, Next, Previous i taster za svaku stranicu koja postoji.

Iz ovoga vidimo da je RestaurantListComponent potrebno implementirati pre PriceComponent i pre PaginationComponent. Postavlja se samo pitanje da li PriceComponent i PaginationComponent međusobno zavise, odnosno da li treba prvo implementirati PriceComponent, PaginationComponent ili je sve jedno. Prethodna analiza nam govori da su ove dve komponente međusobno nezavisne, te nam je sve jedno koju ćemo prvo implementirati.

Treba napomenuti da se i Pagination i Price komponente mogu implementirati pre RestaurantList komponente i da se mogu i testirati. Međutim, testiranje bez prikaza je ipak nešto složeniji proces i za početak je bolje ići jednostavnijim metodom, a to je da se posmatra ispravnost prikaza.

Iz prethodne diskusije možemo izdvojiti dva redosleda implementiranja komponenti, koji su podjednako validni. Prvi je:

1. RestaurantsComponent;
2. RestaurantListComponent;
3. RestaurantItemComponent;
4. RatingComponent.
5. PaginationComponent/PriceComponent;
6. PriceComponent/PaginationComponent.

Drugi je:

1. RestaurantsComponent;
2. RestaurantListComponent;
3. PaginationComponent/PriceComponent;
4. PriceComponent/PaginationComponent;
5. RestaurantItemComponent;
6. RatingComponent.

Dakle, imamo opciju da prvo implementiramo prikaz liste restorana u potpunosti, sve sa stavkom restorana i komponentom za ocenu i cenu ili da prvo implementiramo manipulisanje liste u potpunosti (putem straničenja i sortiranja) pa da onda završimo sa razrađivanjem samog prikaza. Obe opcije su validne i stvar je ličnog izbora.

Na samom kraju se postavlja pitanje u kom momentu treba implementirati RestaurantService, odnosno komunikaciju sa serverom. U opštem slučaju je bolje da se komunikacija sa serverom uradi odmah u startu, kada je poenta neke grupe komponenti da manipulišu sa podacima koji dolaze sa servera, kao što je to ovde slučaj.

Organizacija koda u pakete

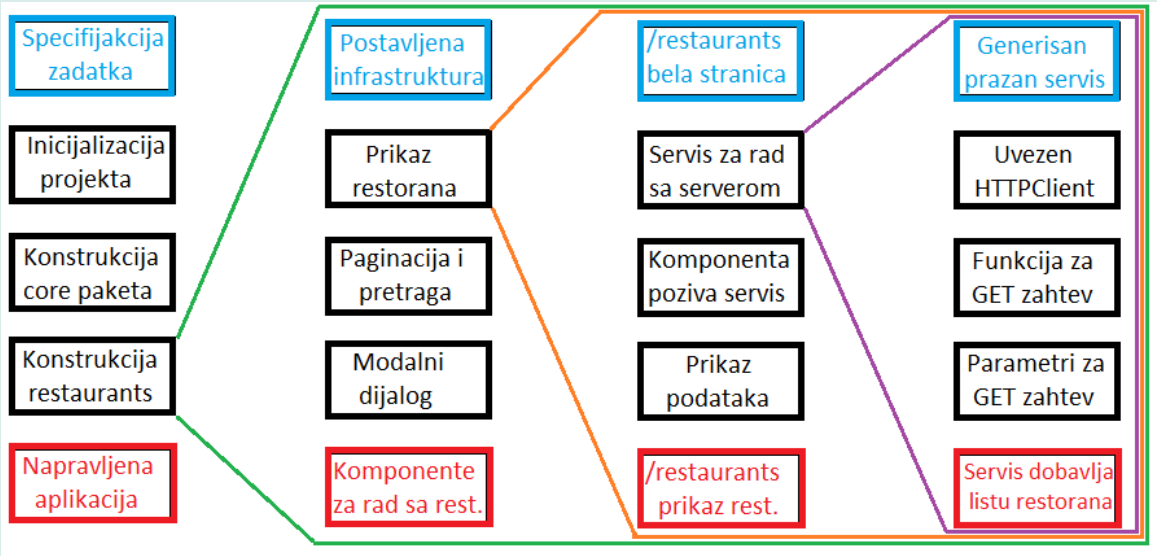
Do sada smo sve komponente generisali u core paketu. Postavlja se pitanje kako dalje organizovati kod, odnosno gde postavljati komponente i servise koji rade sa restoranima. Treba organizovati kod tako da se svako može lako snaći u njemu. Iako sada pišete kod za sebe i samo vi i predavač ga gledate, u firmi ćete sarađivati sa desetinama različitih programera, na projektima koje ćete razvijati mesecima, ako ne i godinama. Ako svako ima svoj stil paketiranja (koji još dodatno menja sa vremena na vreme), taj nedostatak konzistentnosti će značajno otežavati razvoj kako teče vreme. Ovaj problem se rešava tako što se uvode koderske prakse na nivou tima ili firme, kojih se treba pridržavati. *Angular style guide* nudi smernice po pitanju organizacije koda: https://angular.io/guide/styleguide#application-structure-and-ngmodules.

Napomena: Do sada nisu komentarisani zahtevi za pretragu po kuhinji i otvaranje modalnog dijaloga. Razlog iza ovoga jeste što je ovo „nestandardni“ zahtev, odnosno nešto što nismo sretali do sada. Svaki zadatak će imati nekoliko nestandardnih zahteva, gde će polaznik imati priliku da pokaže koliko se dobro snalazi sa novim problemima. Nezavisno od vašeg iskustva, preporuka je da prvo rešite sve što ste procenili da je „standardno“, s’ obzirom da ćete se sa time lakše izboriti, te da ostavite nestandardne zahteve za kraj.

Implementacija liste restorana

Algoritamsko razmišljanje

Kroz čitav proces izrade aplikacije služimo se nečim što nazivamo algoritamsko razmišljanje. Ovo je esencijalna veština koju svaki programer mora da usavrši, nezavisno od tehnologije sa kojom radi. Algoritamsko razmišljanje podrazumeva višestruko dekomponovanje problema na sve sitniji i sitniji niz koraka kako bi se rešio neki problem. Dakle, potrebno je definisati **početnu tačku od koje krećemo** (npr. specifikacija projektnog zadatka), **cilj koji želimo da ostvarimo** (npr. implementirana aplikacija koja ispunjava specifikaciju) i **korake koji su potrebni** da se stigne od početne tačke do cilja. Algoritamskim razmišljanjem smo se služili kroz ceo dokument i nastavićemo. Ilustracija ovog principa je prikazana na slici 4, gde su plavom bojom obeležene početne tačke, crnom koraci, a crvenom cilj.



Slika 4 Razlaganje problema algoritamskim razmišljanjem

Prva kolona posmatra rešavanje kompleksnog problema, što je *Dining Room* zadatak. Prva dva koraka, inicijalizacija projekta i konstrukcija core paketa su razloženi u prvoj sekciji ovog dokumenta. Treći korak je dalje razložen u drugoj koloni, gde su predstavljeni koraci rešavanja kompletne komponente za rad sa restoranima. Prva dva koraka, prikaz restorana i paginacija i pretraga su tema ovog poglavlja. Dalje, u trećoj koloni razlažemo detaljnije problem prikaza restorana, koji je značajno jednostavniji problem od onog u prvoj koloni. Najzad, u četvrtoj koloni razlažemo problem kreiranja servisa za rad sa serverom, gde identifikovane korake možemo direktno da mapiramo na programske instrukcije.

Šabloni u programiranju

Programiranje je prepuno šablona. Kako bude napredovala vaša programerska karijera sve više ćete shvatati da pišete kod koji ste već ranije pisali, sa sitnijim izmenama. Upravo zbog ovoga je vežbanje kodiranja veoma bitno, pošto će iskusni programer mnogo brže prepoznati koji kod treba da iskoristi da bi rešio određen problem, odnosno koji šablon da primeni.

Jedan takav šablon, koji vam je verovatno zapao za oko, jeste komunikacija sa serverom. Bilo da je aplikacija koja radi sa vinima, kućnim ljubimcima, ili restoranima, server koji smo koristili na sličan način pruža sve te podatke. Iako će se serveri sa kojim budete radili razlikovati u određenim aspektima, način na koji će vaša klijentska aplikacija interagovati sa serverom će, u opštem slučaju, biti veoma bliska pristupu koji ste kod nas videli. Ovaj šablon podrazumeva:

- **Model klasu**, koja predstavlja entitet koji dobijate od servera (npr. Wine, Restaurant);
- **Klasu za odgovor**, koja modeluje tačne podatke koji se dobijaju od servera (npr. objekat sa results listom i count poljem);
- **Sam servis**, koji će sadržati sve prevdiđene metode za komunikaciju sa serverom.

Servis će sadržati sve metode koje zadatak zahteva za komunikaciju sa serverom. Metoda će, u zavisnosti od zadatka, prihvatati određene dodatne parametre. Vi, kao razvijač *front-end* aplikacija, dobijate specifikaciju *back-end* servera, koja vam govori šta je API servera, odnosno koje metode možete da pozovete na koji način i sa kojim parametrima.

S' obzirom da je zahtev za dobavaljanja restorana *Dining Room* servera veoma sličan zahtevu za dobavljanja vina *Wine Cellar* servera, brže je preuzeti metodu iz tog servisa i prilagoditi je potrebama *Dining Room* zadatka. Međutim, ovde **treba biti pažljiv**. Iako kopiranje starog koda štedi vreme, slepo kopiranje bez razumevanja će samo dovesti do grešaka koje nećete uspeti zakrpiti. Zbog toga je potrebno imati svest o tome šta tačno radite, odnosno koji šablon koda primenjujete.

Pretraga po kuhinji

Poslednja stavka koja je ostala da se implementira jeste pretraga po kuhinji, koja se realizuje klikom na različite stavke navigacione trake. Da bi rešili ovaj korak potrebno je, kao i uvek, razmotriti šta su koraci koji se dešavaju kako bi mogli da razložimo problem na sitnije probleme.

Korisnik klikom na odabranu stavku navigacione trake, na primer *German* treba da dobije prikaz u RestaurantList komponenti samo onih restorana čija kuhinja (**cuisine**) ima vrednost **german**.

S' obzirom da se SideBar komponenta bavi rutiranjem, putem kog se učitavaju različite komponente, bilo bi dobro da kroz rutiranje objasnimo Restaurants komponenti da želimo restorane određene kuhinje. Ideja je da:

1. Klikom na određenu kuhinju u navigacionoj traci odemo na URL koji označava da hoćemo da vidimo restorane date kuhinje (npr. klikom na *German* URL adresa postane *http://localhost:4200/restaurants/german*);
2. Upotrebom parametra rutiranja izvučemo informaciju koja je kuhinja u pitanju u Restaurants komponenti, odakle prosleđujemo servisu taj podatak putem parametra.

Modifikujemo putanju za RestaurantsComponent, koristeći sledeći kod:

```
{ path: 'restaurants/:cuisine', component: RestaurantsComponent }
```

Što se tiče Sidebar komponente, svaki od linkova ka kuhinjama treba da gađa putanju za datu kuhinju (npr. */restaurants/german*, */restaurants/indian*, itd.).

Da bi mogli da iskoristimo parametar putanje **cuisine** u Restaurants komponenti potrebno je da injektujemo **ActivatedRoute** servis u našu komponentu. Pre nego što nastavite dalje da analizirate ovaj dokument savetujemo da potražite dokumentaciju koja opisuje ovaj servis kako bi se detaljnije upoznali sa njim. Ovo istraživanje ima dvostruku svrhu, da se bolje upoznate sa ActivatedRoute servisom, ali i da vežbate veštinu čitanja i snalaženja sa dokumentacijom.

Čitanje dokumentacije

Čitanje dokumentacije je esencijalna veština bez koje je nemoguće razvijati softver u industriji. Efikasno čitanje dokumentacije je veština koja se trenira. Što više budete čitali o određenoj tehnologiji to ćete je bolje upoznati, ali i pored toga ćete pasivno razvijati svoju sposobnost čitanja i snalaženja sa dokumentacijom, tako da ćete lakše savladati narednu tehnologiju. Poznavanje tehnologije je veština koja je vredna dok je tehnologija aktuelna, no veština istraživanja je nešto što će uvek biti vredno. Ulažite u ovu veštinu što više možete.

U Restaurants kontroleru postavljamo sledeći kod na mesto poziva za getAll funkciju servisa:

```
this.route.params.subscribe( newParams => {
  let cuisine = newParams['cuisine'] == 'all' ? '' : newParams['cuisine'];
  this.parameters.filter.cuisine = cuisine;
  ...
```

Ideja sa prethodnim kodom jeste da kada korisnik ode na *http://localhost:4200/restaurants/all* dobije prikaz svih restorana. Serveru, u tom slučaju, treba da pošaljemo praznu kuhinju. Ovo je opet specifičnost servera što je dokumentovano za vas kao front-end programere u sklopu specirfikacije server API-a. Sada nam je omogućena pretraga po kuhinji.

Rad sa spoljnim bibliotekama

Kroz ovaj zadatak smo spomenuli nekoliko esencijalnih veština koje svaki programer, nezavisno od tehnologije sa kojom radi, treba da poseduje i da učestalo razvija. Fundamentalna veština predstavlja **algoritamsko razmišljanje**, gde se problem posmatra kao skup manjih potproblema, tako da se svaki problem razlaže na niz sitnih koraka koje treba realizovati kako bi se problem rešio. **Čitanje i snalažanje sa dokumentacijom** tehnologije u kojoj radite je takođe ključna veština, kao i **metodološki pristup rešavanju grešaka** i **poznavanje i upotreba šablonskih rešenja**. Iako ne esencijalna, veština **dobre organzacije koda i pisanja čistog koda** predstavlja nešto što svaki programer treba da učestalo razvija.

Sve prethodno navedene veštine se razvijaju upornim i kontinualnim treningom. Kako se ove veštine razvijaju pasivno, možda nećete primetiti da napredujete na ovom polju na kraćim stazama, no ako pogledate u periodu od pola godine rada koliko ste poznavali neku oblast i snalazili se sa njom na početku tog perioda, a koliko na kraju, videćete da tu zaista postoji velik skok. Poslednja veština koju ćemo diskutovati u ovom dokumentu predstavlja **sposobnost snalaženja sa spoljnim bibliotekama** ili, generalnije, snalaženje sa tuđim kodom.

Jedna od glavnih karakteristika Angular radnog okvira jeste upotreba komponenti. Za sada smo koristili isključivo komponente da prikažemo bilo kakav HTML na stranici. Prednosti ovakve organizacije koda su mnogostruke, od bolje organizacije koda do lakše organizacije posla implementacije koda. Jedna od najznačnijih prednosti predstavlja mogućnost ponovnog iskorišćavanja koda i ovde postoje dva tipa upotrebe komponente:

- Upotreba jedne komponente na više mesta u isto projektu;
- Upotreba jedne komponente na više mesta u više projekata.

Jedan klasičan primer komponente koju smo viđali u našim zadacima (a i na velikom broju veb-sajtova) jeste Pagination komponenta. Ovo je komponenta koja se nalazi u gotovo svakoj veb-aplikaciji koja prikazuje neki vid liste podataka.

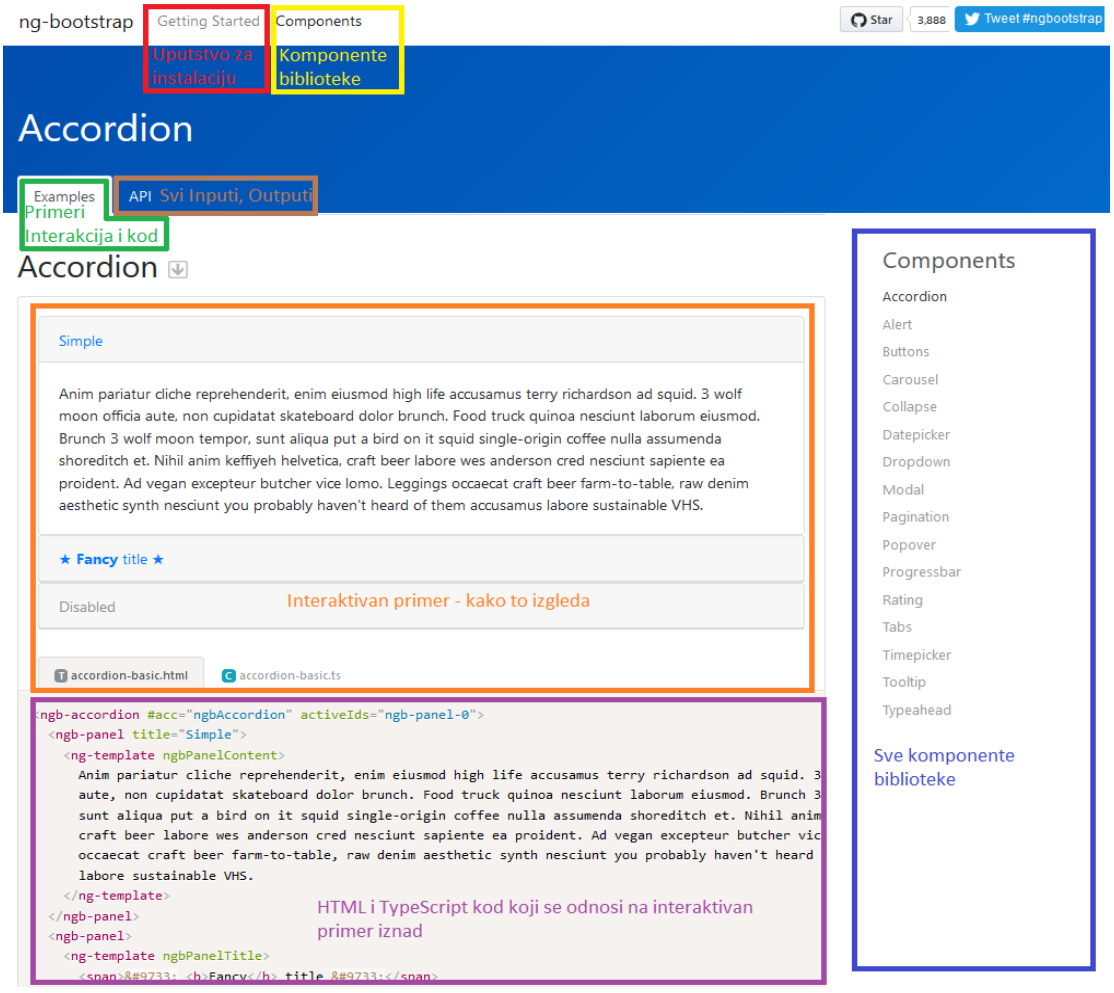
Kada god imamo zahtev da implementiramo komponentu koju smo sretali po internetu, a koristimo popularnu tehnologiju poput Angulara, postoji dobra šansa da je već neko to uradio pre nas i da je negde objavio. Lepota radnog okvira zasnovanog na komponentama jeste što je, u opštem slučaju, ugrađivanje takvih komponenti dosta jednostavno, mnogo jednostavnije nego da sami ručno implementiramo datu komponentu.

Poslednji deo zadatka koji još uvek nismo implementirali predstavlja modalni dijalog. Klikom na određen restoran treba da se otvori dijalog koji pokazuje detalje o datom restoranu, kao i njegov meni. Implementiranje modalnog dijaloga bi zahtevalo da se igramo sa CSS i JavaScript kodom, što bi potencijalno oduzelo više sati. Međutim, ako pronađemo komponentu za lako kreiranje modalnog dijaloga, ovaj posao možemo realizovati u roku od najviše sat vreme. Tokom ovog vremena:

1. Pronalazimo biblioteka koja sadrži datu komponentu;
2. Proučavamo način instalacije i upotrebe biblioteke, što se svede na izučavanje dokumentacije;
3. Instaliramo i zatim iskoristimo biblioteku tako da rešimo naš zadatak.

Samo pronalaženje prikladne biblioteke je veština koju treba usavršiti. Prilikom pretrage biblioteka (npr. putem Google servisa) treba voditi računa o njenom kvalitetu, odnosno o posvećenosti tvorca biblioteke. Ako neka biblioteka ima dobru, ažurnu dokumentaciju na GitHub-u (ili još bolje ima sopstven sajt sa svim informacijama) to je dobar znak da je biblioteka kvalitetna. Za naš slučaj nam je potrebna biblioteka koja nudi modalni dijalog. Upit poput „*Angular 4 modal dialog*“ bi bio dobar za početak. Ono što ćete primetiti kucanjem prethodnog upita u Google jeste da postoji više biblioteka koje nude ovu komponentu. Tako se jedna zasniva na *material design* CSS bibliotekama, dok postoji i jedna koja koristi Bootstrap CSS, te ćemo baš nju i koristiti. U pitanju je *ng-bootstrap*: https://ng-bootstrap.github.io/#/home.

Na vrhu veb-sajta ove biblioteke pronalazimo link *Getting Started*. Na ovom linku ćemo pronaći uputstvo kako da instaliramo ovu biblioteku. Naredni link, *Components*, sadrži interaktivne primere, zajedno sa kodom koji pokazuje kako izgledaju komponente biblioteke, kao i primere koda kako da se iskoristi komponenta u projektu. Svaka komponenta sadrži dva taba, *Examples*, gde se nalaze primeri sa pratećim kodom, i *API*, gde se nalazi formalna specifikacija komponente, sa svim inputima, outputima i funkcijama koje podržava, uz kratak opis svake. Slika 5 prikazuje celine koje su iznad opisane.



Slika 5 Pregled ng-bootstrap veb-sajta

Radi vežbanja čitanja i snalaženja sa dokumentacijom, kao i sa spoljnim bibliotekama, na vama je da implementirate modalni dijalog po uzoru na *Components as content* sekciju kod Modal komponente.