Arhitektura zasnovana na komponentama

Hajduković, Mateo

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni preddiplomski studij Matematika i računarstvo

Arhitektura zasnovana na komponentama

Završni rad

Mentor:

izv. prof. Domagoj Matijević

Komentor:

dr. sc. Jurica Maltar

Kandidat:

Mateo Hajduković

Osijek, 2024

Sadržaj

1 | Uvod

Računarstvo je danas jedna od najbrže rastućih grana znanosti te se slijedom toga kontinuirano traže novi i efikasniji pristupi problemima i samom razvijanju konačnog softverskog proizvoda.

Jedan od tih novih pristupa ujedno je i Arhitektura zasnovana na komponentama (u daljnjem tekstu: CBA). CBA je vrlo efikasan pristup izgradnji softvera temeljen na, kao što i sam naziv govori, komponentama. Komponenta je većinom samostalna jedinica u kodu koja rješava svoj dio problema, a ujedno je i ponovno upotrebljiva, što bi značilo da kombinacijom komponenti dolazimo do završnog proizvoda.

Komponente pružaju viši stupanj apstrakcije dijeleći problem na pod-probleme. U ovom radu naglasak je na konceptima CBA, njegovim prednostima i manama te na korištenju u današnjoj industriji.

1.1 Glavni koncepti CBA

Kako bismo bolje razumjeli CBA, poslužimo se slikovitim primjerom opisa kuhinje. Ta kuhinja ima svoje elemente kao što su primjerice hladnjak, pećnica, sudoper i drugi. Svaki od tih elemenata ima svoju individualnu funkciju koja je u konačnici dio većeg sustava – kuhinje. Također, većina kuhinja ima svoje standardne dimenzije za elemente, što olakšava mijenjanje istih. Ako se samo hladnjak pokvari, ne moramo mijenjati cijelu kuhinju već samo hladnjak.

CBA se vrti oko ideje izgradnje softverskih sustava kombiniranjem samostalnih i neovisnih jedinica koje nazivamo komponente. Komponente su modularne, što znači da svaka sadrži određeni dio funkcionalnosti s jasnom naznakom kako druge komponente mogu komunicirati s njom. Baš ta modularnost omogućava razdvajanje jednog velikog i kompleksnog problema na manje i jednostavnije.

1.2 Zašto CBA?

Potreba za izgradnjom CBA proizašla je iz nedostataka tradicionalne monolitne arhitekture softvera. U monolitnoj arhitekturi, softverski sustav izgrađen je kao jedna velika nedjeljiva cjelina u kojoj su svi dijelovi softvera međusobno povezani i integrirani. To znači da su sve funkcionalnosti softvera smještene unutar jednog čvrsto povezanog koda. Ovakav pristup može dovesti do problema jer

kako softver raste i postaje sve složeniji, održavanje i dodavanje novih funkcionalnosti postaje sve teže. Svaka promjena u jednom dijelu koda može imati nepredvidive posljedice na druge dijelove softvera, što otežava razvoj, testiranje i skaliranje softvera. Nasuprot tome, CBA omogućava podjelu softvera na manje, nezavisne komponente, što ubrzava razvoj i shodno tome dolazi do jednostavnijeg održavanja koda i lakšeg shvaćanje cijelog softvera.

Treba napomenuti da CBA softver dijeljenjem na komponente potiče njihovu ponovnu upotrebu, što bi značilo da ako neku komponentu moramo iskoristiti u više dijelova koda, ne moramo pisati iznova taj kod već samo proslijediti već napisanu komponentu.

Također, u današnje vrijeme razvoja softverskih proizvoda, velike firme imaju jako puno zaposlenika koji u isto vrijeme rade na istim projektima. Izumom CBA taj rad postao je uvelike jednostavniji jer zaposlenici mogu puno lakše podijeliti softver na komponente i svatko zasebno raditi te na kraju doći do konačnog proizvoda. Tu se može spomenuti i adaptacija na nove potrebe – kako u današnje vrijeme postoji sve više cloud proizvoda, tako je postalo iznimno važno da se konačni softver može podijeliti u više jedinica u kojem će svaka od njih odrađivati svoj dio posla, u ovom slučaju mikro-servisa.

1.3 Izazovi CBA

CBA nam olakšava rad i nudi puno prednosti kroz razvijanje proizvoda, no također donosi i izazove koji nisu zanemarivi.

Jedan od glavnih izazova je integracija komponenti, posebno kada se komponente razvijaju neovisno, često od strane različitih timova, ali i od nekog drugog dobavljača. Takve komponente mogu imati različite principe dizajna, standarde koda ili drugačiju komunikaciju s ostalim komponentama, što dovodi do značajnih problema pri integraciji istih. Osiguravanje da komponente dobro rade samostalno i jedne s drugima zahtjeva rigorozno testiranje i dobro definiran princip razvoja komponenti.

Izazovi su također i performanse tog softverskog proizvoda. Iako svaka komponenta u CBA komunicira putem dobro definiranih sučelja i potiče modularnost, također može doći do većeg troška obrade podataka te kašnjenja. Takve male pogreške mogu se akumulirati, posebice u komponentama koje često moraju komunicirati jedne s drugima te imaju složenije funkcionalnosti. Programeri u tom slučaju moraju uravnotežiti tu potrebu za modularnošću komponenti s potrebom za učinkovitijom komunikacijom i smanjivanjem troškova obrade podataka.

2 | CBA u današnjoj industriji

Arhitektura temeljna na komponentama temelj je za razvoj modernog softvera, posebno kao odgovor na sve veću potrebu industrije za brzorastućim, održivim i fleksibilnim softverima. Dok se CBA najviše koristi u razvoju web aplikacija, ne smiju se zanemariti ni enterprise softveri i mikro servisi.

2.1 Web development

Moderni web development danas se oslanja na takozvane "frameworkove" koji se temelje na CBA, primjerice React, Angular i Vue. Oni omogućuju programerima da jednostavno izrade korisničko sučelje putem komponenti koje se mogu ponovno iskoristiti. Svaka komponenta predstavlja dio korisničkog sučelja poput gumba, obrasca ili vizualizacije podataka, koji se mogu ponovno koristiti u različitim dijelovima aplikacije ili čak u različitim projektima. Danas je i programerska zajednica velika pa programeri mogu preuzimati već unaprijed napravljene komponente od drugih programera i integrirati ih u svoj softverski proizvod.

2.2 Enterprise softveri

Enterprise softver često podrazumijeva integraciju usluga ili komponenti trećih strana koje su specijalizirane za određene zadatke. Umjesto da sami razvijamo sve funkcionalnosti od nule, često koristimo specijalizirane komponente koje nude vanjski dobavljači. Takvi primjeri uključuju softvere za obradu plaćanja ili provjeru autentičnosti korisnika, koji su dizajnirani da se lako integriraju u naš softverski proizvod putem vanjske komponente. Ova praksa omogućava programerima brzu i jednostavnu integraciju složenih funkcionalnosti bez potrebe za ulaganjem značajnih resursa u razvoj tih specifičnih dijelova softvera.

2.3 Mikro servisi

Arhitektura mikro servisa, koja je postala dominantan obrazac u današnjoj industriji, također se temelji na CBA. Mikro servisi dopuštaju (kao što im i ime govori) da podijelimo zadaće softvera u malene servise koji su neovisni o našem softveru. Svaki mikro servis obično je smješten na cloudu i s njega komunicira sa softverom. Korištenje mikro servisa omogućava programerima da imaju različiti tempo

razvijanja mikro servisa i konačnog softvera, kao i ponovnu upotrebljivost u drugim softverima.

3 | CBA na primjeru aplikacije

Uporabu CBA možemo prikazati na primjeru male aplikacije za upravljanje korisnicima. Aplikaciju pišemo u JavaScriptu - uglavnom prvom izboru što se tiče web developmenta. Nećemo pisati u čistom JavaScriptu, već u Reactu čija se inačica JavaScripta zove JSX, a koju je potrebno prevesti u čisti JavaScript, HTML i CSS kako bi preglednik bio u stanju interpretirati taj kod. React je biblioteka za kreiranje korisničkih sučelja i ujedno jedan od najpopularnijih izbora za izradu istih.

3.1 Problem

Želimo napraviti malo sučelje za ispis svih korisnika koji koriste neku uslugu - kod korisnika, uz njegovo ime, trebaju biti prikazane i njegove godine. Također želimo imati mogućnost filtriranja korisnika prema imenu i dodavanja novog korisnika.

3.2 Implementacija

3.2.1 Korisnik

Krenimo od prvog dijela problema - moramo prikazati korisnike, stoga želimo implementirati prikaz jednog korisnika kojeg ćemo kasnije prikazivati više puta. To ćemo svrstati u jednu komponentu. Komponenta će primiti podatke od jednog korisnika.

```
1 const User = ({ user }) => {
2   return (
3     <li>
4       {user.name}, Godine: {user.age}
5     </li>
6   );
7 };
8
9 export default User;
```

3.2.2 Lista korisnika

Nadalje, želimo prikazati više korisnika odjednom - kreirajmo komponentu koja će primiti sve korisnike u jednoj listi, iterirati kroz tu listu i prikazati informacije putem naše prethodno kreirane **User** komponente.

```
import User from './User';

const UserList = ({ users }) => {
  return (
    <ul>
      {users.map((user, index) => (
        <User key={index} user={user} />
      ))}
    </ul>
  );
};

export default UserList;
```

.map() iterira kroz svaki podatak u listi te ga mapira po željenoj funkciji.

3.2.3 Spajanje komponenti

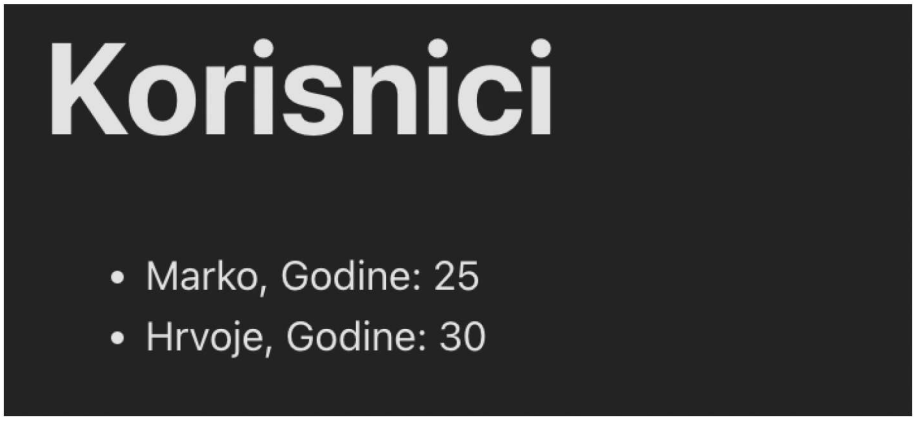
Dosada smo napisali komponentu za prikaz informacija o korisniku te komponentu za prikaz svih korisnika iz liste. Sada moramo napraviti prikaz koji će objediniti te komponente u smislenu cjelinu. Stoga pravimo komponentu koju ćemo nazvati **App**, jer će ona biti glavna jedinica naše aplikacije. Također, kreiramo listu korisnika koja će sadržavati informacije o korisniku (ime i godine) te je spremamo u varijablu kako bismo je mogli proslijediti drugim komponentama.

```
import { useState } from 'react';
import UserList from './UserList';

const App1 = () => {
  const [users] = useState([
    { name: 'Marko', age: 25 },
    { name: 'Hrvoje', age: 30 },
  ]);

  return (
    <div className="app">
      <h1>Korisnici</h1>
      <UserList users={users} />
    </div>
  );
};

export default App1;
```



Slika 3.1: Izlistavanje korisnika

Naša je aplikacija prikazana na Slici 3.1. Već možemo primijetiti principe CBA u dosadašnjem kodu. Napisali smo kod za prikaz korisnikovih imena i godina - i njega možemo koristiti bilo gdje u aplikaciji. Kada god odlučimo promijeniti nešto oko tog prikaza, možemo samo otići u tu komponentu i promijeniti to na svim mjestima u kodu. Također, ako imamo nekakvih problema oko prikaza korisnikovih informacija, znamo da je problem vrlo vjerojatno u toj jednoj komponenti koja se bavi prikazivanjem korisnikovih informacija. Još jedan od principa koji primjećujemo jest odgovornost komponenti - komponenta **User** bavi se samo prikazivanjem informacija korisnika i ničim drugim, isto kao i **UserList** - prima listu korisnika i mapira je prema **User** komponenti.

3.2.4 Dodavanje novog korisnika

Nadalje, želimo funkcionalnost dodavanja novog korisnika. Trebaju nam dva polja za unos informacija (ime i godine) te gumb za potvrdu. Potrebna nam je varijabla koja će pratiti informaciju koju unosimo u polja i funkcija koja će dodati novog korisnika u listu.

```
1  import { useState } from 'react';
2
3  const AddUser = ({ onAddUser }) => {
4    const [newUser, setNewUser] = useState({ name: '', age: '' });
5
6    const handleAddUser = () => {
7      if (newUser.name && newUser.age) {
8        onAddUser({ name: newUser.name, age: parseInt(newUser.age) })
       ;
9        setNewUser({ name: '', age: '' });
10     }
11   };
12
13   return (
14     <div>
15       <input
16         type="text"
```

```
17          value={newUser.name}
18          onChange={(e) => setNewUser({ ...newUser, name: e.target.
    value })}
19          placeholder="Ime"
20        />
21        <input
22          type="number"
23          value={newUser.age}
24          onChange={(e) => setNewUser({ ...newUser, age: e.target.
    value })}
25          placeholder="Godine"
26          style={{ margin: '0 10px' }}
27        />
28        <button onClick={handleAddUser}>Dodaj korisnika</button>
29      </div>
30    );
31  };
32
33  export default AddUser;
```

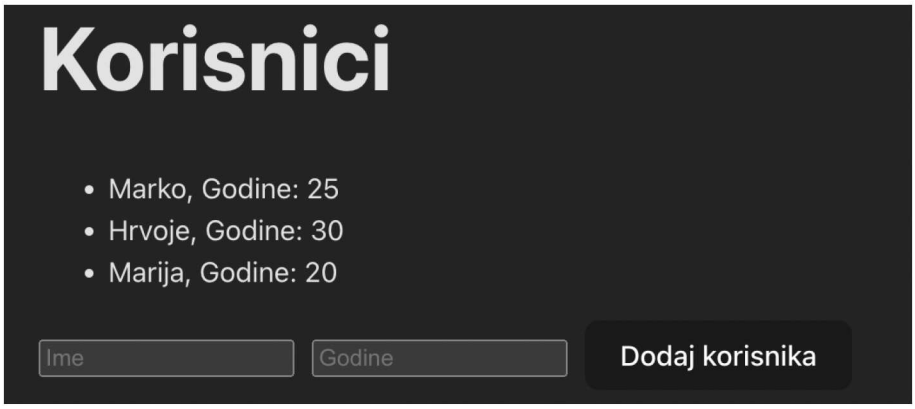
Kao što vidimo, komponenta prima funkciju **onAddUser** koja će na kraju dodati tog korisnika u listu. Razlog zašto tu funkciju nismo pisali u ovom kodu jest to što ne želimo da ova komponenta brine o prijašnjim podacima ili da postane previše ovisna o ostalim komponentama. Stoga joj samo prosljeđujemo funkciju koju kasnije možemo mijenjati ili dodavati logiku bez mijenjanja izvorne komponente.

Funkciju pišemo u komponenti koja će biti zadužena za podatke liste korisnika, a to bi bila naša glavna komponenta **App**. Kod glavne komponente opisan je algoritmom 3.1. dok je izgled aplikacije prikazan na slici 3.2.

```
1  import { useState } from 'react';
2  import UserList from './UserList';
3  import AddUser from './AddUser';
4
5  const App2 = () => {
6    const [users, setUsers] = useState([
7      { name: 'Marko', age: 25 },
8      { name: 'Hrvoje', age: 30 },
9    ]);
10
11   const addUser = (newUser) => {
12     setUsers([...users, newUser]);
13   };
14
15   return (
16     <div className="app">
17       <h1>Korisnici</h1>
18       <UserList users={users} />
19       <AddUser onAddUser={addUser} />
20     </div>
21   );
22 };
23
```

```
export default App2;
```

Algoritam 3.1: App.jsx



Slika 3.2: Dodavanje novog korisnika

Ovdje također možemo vidjeti princip odvojene odgovornosti gdje komponenta za dodavanje novog korisnika ne brine o tome kako izgleda lista niti kako se informacije dodaju u listu - ona samo brine o slanju i spremanju informacija novog korisnika.

3.2.5 Filtriranje korisnika

Zadnja funkcionalnost koju želimo implementirati jest filtriranje korisnika prema imenu. Potrebno nam je jedno polje u koje unosimo vrijednost filtra te funkcija koja će filtrirati rezultate.

```
import { useState } from 'react';

const FilterUser = ({ onFilter }) => {
  const [filter, setFilter] = useState('');

  const handleFilter = (e) => {
    setFilter(e.target.value);
    onFilter(e.target.value);
  };

  return (
    <div>
      <input
        type="text"
        value={filter}
        onChange={handleFilter}
        placeholder="Pretra i korisnike"
      />
    </div>
  );
};
```

```
22
23 export default FilterUser;
```

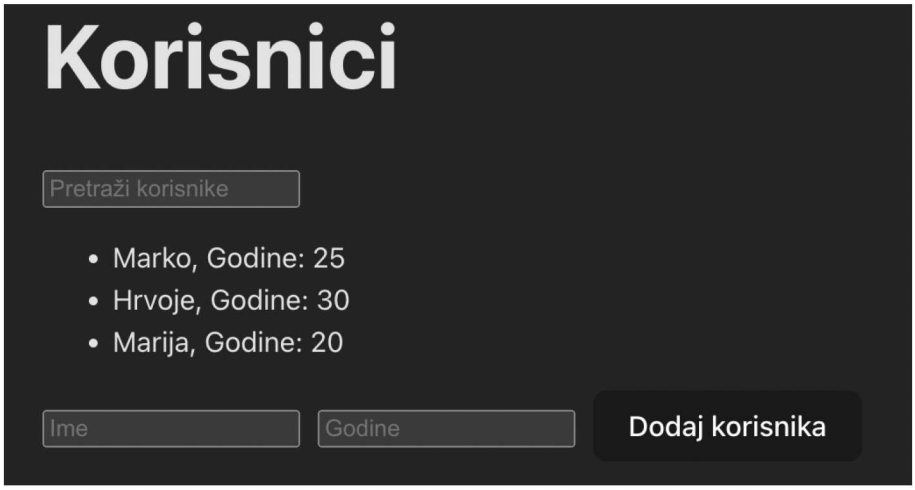
Opet vidimo da komponenta **FilterUser** ne mora brinuti o tome kako ćemo filtrirati korisnike već se samo mora pobrinuti da vrijednost filtra dođe do komponente koja se bavi prikazom korisnika.

Kod glavne komponente opisan je algoritmom 3.2.

```
1  import { useState } from 'react';
2  import UserList from './UserList';
3  import AddUser from './AddUser';
4  import FilterUser from './FilterUser';
5
6  const App = () => {
7    const [users, setUsers] = useState([
8      { name: 'Marko', age: 25 },
9      { name: 'Hrvoje', age: 30 },
10   ]);
11   const [filter, setFilter] = useState('');
12
13   const addUser = (newUser) => {
14     setUsers([...users, newUser]);
15   };
16
17   const filterUsers = (searchTerm) => {
18     setFilter(searchTerm);
19   };
20
21   const filteredUsers = users.filter((user) =>
22     user.name.toLowerCase().includes(filter.toLowerCase())
23   );
24
25   return (
26     <div className="app">
27       <h1>Korisnici</h1>
28       <FilterUser onFilter={filterUsers} />
29       <UserList users={filteredUsers} />
30       <AddUser onAddUser={addUser} />
31     </div>
32   );
33 };
34
35 export default App;
```

Algoritam 3.2: App.jsx

Dodali smo varijablu koja prati vrijednost koju vraća **FilterUser** komponenta, te varijablu koja filtrira naše korisnike prema toj vrijednosti, a te filtrirane korisnike prosljeđujemo **UserLists** komponenti. Stoga je naša **App** komponenta preuzela odgovornost za manipuliranje listom korisnika i spajanjem svih komponenti.



Slika 3.3: Konačna aplikacija

3.2.6 Konačna aplikacija

Naša aplikacija, nakon implementacije svih funkcionalnosti, prikazana je na slici 3.3. U aplikaciji možemo vidjeti principe CBA:

- Odvajanje odgovornosti - komponenta **UserList** bila je zadužena samo za prikaz korisnika, **AddUser** samo za dodavanje korisnika, **FilterUser** samo za filtriranje korisnika.
- Ponovna upotrebljivost - komponente poput **User** ili **UserList** možemo upotrijebiti na bilo kojem drugom mjestu u aplikaciji dok god je struktura podataka ista.
- Održavanje i nadograđivanje - premda je logika aplikacije podijeljena u puno malih dijelova, vrlo je lako nadograđivati i održavati aplikaciju bez previše promjena. Na primjer, ako želimo promijeniti nešto u **User** komponenti, ne moramo dirati nijednu drugu komponentu osim te.
- Komunikacija komponenti - kao što smo vidjeli, komponente komuniciraju putem proslijeđenih funkcija ili varijabli, čime postižemo odvajanje odgovornosti.
- Testiranje - testiranje aplikacije podijeljene u komponente je iznimno lako jer svaka komponenta može imati svoj test - **Unit testing**, a ako želimo testirati kako komponente rade jedne s drugima u cjelini, koristimo **E2E testing**.

4 | Usporedba React i Angular komponenti

React i Angular, iako su danas među glavnim izborima za izradu web aplikacija, imaju nešto drugačije pristupe kada dolazi do same implementacije aplikacije. React je vrlo fleksibilan i jednostavan, što ga čini izvrsnim izborom za manje web aplikacije. S druge strane, Angular sa svojom jasnom strukturom i bogatim ekosustavom puno je bolji izbor za veće i kompleksnije aplikacije. Kako bismo bolje shvatili njihove razlike implementirat ćemo jednostavnu aplikaciju brojača. Kada kliknemo gumb "Increase" želimo povećati vrijednost brojača, a kada kliknemo "Decrease" želimo smanjiti vrijednost.

4.1 React brojač

```
1  import { useState } from 'react';
2
3  const Counter = () => {
4    const [count, setCount] = useState(0);
5
6    return (
7      <div>
8        <h1>Counter: {count}</h1>
9        <button onClick={() => setCount(count + 1)}>Increase</button>
10       <button onClick={() => setCount(count - 1)}>Decrease</button>
11     </div>
12   );
13 }
```

U Reactu koristimo funkcijske komponente, što samo ime sugerira – komponenta je zapravo funkcija koja vraća kombinaciju HTML-a i JSX-a. React također sadrži posebne funkcije nazvane "hooks", koje služe za upravljanje stanjem varijabli i životnim ciklusima komponenti. Praćenje vrijednosti varijable count omogućeno je putem useState hooka. useState omogućuje upravljanje stanjem varijabli unutar funkcijskih komponenti te vraća niz s dvije vrijednosti – trenutnu vrijednost varijable i funkciju koja postavlja novu vrijednost. Ovo je najjednostavniji način upravljanja stanjem varijabli u Reactu. Dodatno, potrebno je osigurati da se setter pozove pri kliku na gumb. Za upravljanje klikom koristimo onClick event

kojem prosljeđujemo funkciju koja se aktivira prilikom klika.

4.2 Angular brojač

counter.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-counter',
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.css']
})
export class CounterComponent {
  count = 0;

  increase() {
    this.count++;
  }

  decrease() {
    this.count--;
  }
}
```

counter.component.html

```
<div>
  <h1>Counter: {{ count }}</h1>
  <button (click)="increase()">Increase</button>
  <button (click)="decrease()">Decrease</button>
</div>
```

Kod Angular komponenti odmah primjećujemo da logiku i prikaz korisničkog sučelja ne možemo kombinirati u jednom dokumentu, već ih razdvajamo u dva. Ovo jasno prikazuje podjelu odgovornosti unutar same komponente. Logika je smještena u klasu unutar TypeScript dokumenta, dok je korisničko sučelje definirano u HTML-u. Stanje varijable prati se u klasi, a Angular koristi dvosmjerno povezivanje podataka, što omogućuje automatsku sinkronizaciju između prikaza i logike. Što se tiče događaja (eng. events), oni su također definirani u HTML atributima, slično kao u Reactu, ali uz nešto drugačiju sintaksu.

5 | Zaključak

Arhitektura zasnovana na komponentama(CBA) vrlo je učinkovita i suvremena metoda razvoja softverskog proizvoda. Zbog mogućnosti izgradnje složenih softvera na modularan i fleksibilan način, CBA donosi ponovnu upotrebljivost koda, lakše održavanje i skaliranje softvera. CBA je nastao kao odgovor na ograničenja monolitnih arhitektura koje su s rastom postale kompleksne i teško održive. Iako CBA ima svoje izazove, poput primjerice problema u integraciji komponenti i smanjenju performansi, upotreba ovog pristupa u današnjoj industriji vrlo je široka, posebice u razvoju web aplikacija, enterprise softvera i u mikro servisima, što jasno pokazuje važnost i učinkovitost CBA. Također kada dolazi do samog izbora okruženja u kojem ćemo razvijati aplikaciju treba dobro proučiti karakteristike koje će naša aplikacija imati, koliko ljudi će raditi na njoj te na kraju pogledati koja tehnologija ima više prednosti za naš proizvod.

Literatura

[1] Blair, G., Coupaye, T. , Stefani, JB., *Component-based architecture: the Fractal initiative.*, Ann. Telecommun. 64, 1–4 (2009).

[2] George Heineman, William Councill, *Component-Based Software Engineering: Putting the Pieces Together.*, 2001.

[3] https://nandbox.com/all-you-need-to-know-about-component-based-architecture/

[4] https://en.wikipedia.org/wiki/Component-based_software_engineering

[5] https://react.dev/

[6] https://angular.dev/

Sažetak

Arhitektura zasnovana na komponentama (CBA) je jako efikasna i suvremena metoda razvoja softverskog proizvoda. Zbog mogućnosti izgradnje složenih softvera na modularan i fleksibilan način CBA donosi ponovnu upotrebljivost koda, lakše održavanje i skaliranje softvera. CBA je nastao kao odgovor na ograničenja monolitnih arhitektura koje su sa rastom postale kompleksne i teško održive. Iako CBA ima svoje izazove - poput problema u integraciji komponenata i smanjenju performansi, njena upotreba u današnjoj industriji je jako široka - posebice u razvoju web aplikacija, enterprise softvera i mikro servisima, jasno pokazuje njenu važnost i učinkovitost.

Ključne riječi

komponenta, arihitektura, softver, modularnost, fleksibilnost, razvoj, programeri, izazovi, održavanje, skaliranje, efikasnost

Component-based architecture

Summary

Component-Based Architecture (CBA) is a highly efficient and modern approach for developing software products that are based on components. Because of its ability to build complex software in a modular and flexible way, CBA brings code reusability, easier maintenance, and scalability of software. CBA was made to address the old monolithic architecture that would become complex and hard to maintain as it grew. Despite CBA having its own challenges—like problems with the integration of components or a decrease in software performance—its use is still widespread in today's industry, especially in the development of web applications, enterprise software, and microservices, clearly showing its importance.

Keywords

component, architecture, software, modularity, flexibility, development, programmer, challenges, maintaining, scalability, efficiency