



## ПРАКТИКУМ

РАЧУНАРСКЕ ВЕЖБЕ



## Садржај

Увод у React. Vite.JS .....	1
Вежба 1: Припрема пројекта .....	2
Вежба 1: Задатак .....	5
Вежба 1: Решење .....	6
Валидација података. Local Storage .....	9
Вежба 2: Методи валидације података .....	10
Вежба 2: Local Storage .....	11
Вежба 2: Задатак .....	12
Вежба 2: Решење .....	13
Клијент-Сервер архитектура. Примена SOLID принципа и чисте архитектуре .....	19
Вежба 3: Клијент-Сервер архитектура .....	20
Вежба 3: SOLID принципи .....	21
Вежба 3: Чиста архитектура .....	22
Вежба 3: Задатак .....	23
Вежба 3: Решење .....	24
Репликација и отпорност на отказе .....	41
Вежба 4: Репликација .....	42
Вежба 4: Отпорност на отказе .....	42
Вежба 4: Vercel .....	43
Вежба 4: Задатак .....	44
Надзор и управљање апликацијом у дистрибуираном систему .....	45
Вежба 5: Deployment апликације на Vercel .....	46
Вежба 5: Управљање евиденцијом догађаја .....	47
Безбедносни механизми: Аутентификација и ауторизација .....	48
Вежба 6: Аутентификација .....	49
Вежба 6: Ауторизација .....	??
Вежба 6: Задатак .....	??
Вежба 6: Решење .....	??
Контрола приступа заснова на улогама .....	??
Вежба 7: Хијерархија корисника система .....	??
Вежба 7: Одређивање и контрола нивоа приступа систему .....	??
Вежба 7: Задатак .....	??
Вежба 7: Решење .....	??



# УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Увод

У области развоја веб апликација, све већи значај имају алати и библиотеке који омогућавају ефикасан, скалабилан и одржив приступ изградњи корисничких интерфејса. **React** представља библиотеку за креирање интерактивних веб апликација кроз компонентно оријентисану архитектуру. Његова основна концептуална предност лежи у могућности да се сложени кориснички интерфејси поделе на **мање, независне компоненте** које комуницирају међусобно и са спољним окружењем. Оваква модуларност не само да поједностављује процес развоја, већ и омогућава лакше одржавање, тестирање и проширење постојећег кода. Поред тога, **React** омогућава ефикасно ажурирање података путем виртуелног модела **Document Object Model**, што значајно унапређује перформансе апликације.

Са друге стране, **Vite.js** представља савремену алатку за конфигурисање и покретање развојног окружења, која се све више користи у пројектима базираним на **JavaScript**-у и његовим библиотекама, као што је **React**. За разлику од традиционалних *build* система, **Vite** користи приступ базиран на **ES модулима** и директно испоруци ресурса прегледачу, што резултира знатно бржим покретањем и освежавањем апликација током развоја.

**Vite** такође омогућава флексибилну конфигурацију и подршку за савремене **front-end** стандарде, што га чини изузетно погодним за пројекте различите сложености. Комбиновањем **React** и **Vite**, добијаја се моћно окружење које подржава брз развој, високе перформансе и **чисту архитектуру**, што је од посебног значаја у контексту модерног веб програмирања.

## Потребан софтвер

- **Visual Studio Code**  
<https://code.visualstudio.com/download>
- **Node.js**  
<https://nodejs.org/en/download>

## Питања

- Шта представља компонента у *React*-у?
- Која је основна предност коришћења *React*-а у односу на класичан *JavaScript*?
- Шта је *JSX* односно *TSX* и која му је сврха у *React*-у?
- Како се разликују функционалне и класне компоненте у *React*-у?
- Која је основна улога *Vite.js* у развоју веб апликација?
- Зашто је *Vite* бржи у односу на традиционалне алате попут *Webpack*-а?
- Шта је *Node Package Manager*?
- Чему служи *package.json*?



# УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Вежба

Неопходно је креирати **нови празан фолдер** на радној површини, а затим отворити **Visual Studio Code**. Из менија одабрати опцију: **File → Open Folder** и одабрати претходно креиран фолдер на радној површини и кликнути на дугме **Select Folder**.

Са леве стране биће видљив празан директоријум и отворен фолдер. Из менија одабрати **Terminal → New Terminal** након чега ће се појавити командни прозор у који је потребно унети следећу команду:

```
npm create vite@latest
```

У случају да се команда покреће први пут, потребно је унети карактер **y (скр. yes)** како би се потврдила инсталација **Vite** пакета. У случају да је команда раније покренута појавиће се следећи испис у терминалу:

```
> npx
> create-vite
|
◆ Project name:
| vite-project
L
```

Уместо **vite-project** потребно је унети име пројекта, име је произвољно, нпр. **odp1** након чега је потребно одабрати са којом библиотеком ћемо радити.

◆ **Select a framework:**

- vanilla
- Vue
- React
- Preact
- Lit
- Svelte
- Solid
- Qwik
- Angular
- Marko
- Others

Стрелицама одаберемо **React** као жељену опцију за наш пројекат.



# УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Вежба

Сада је потребно одабрати да ли ћемо користити подршку за типове или не, наравно бирајмо опцију за подршку за типове односно **TypeScript**:

- ◆ Select a variant:
  - TypeScript
  - TypeScript + SWC
  - JavaScript
  - JavaScript + SWC
  - React Router v7 ↗
  - TanStack Router ↗
  - RedwoodSDK ↗

Након одабира опције креиран је почетни шаблон за веб апликацију уз даља упутства за промену директоријума и инсталирање неопходних пакета.

```
◆ Scaffolding project in
C:\Users\Danijel\ODP\odp1...
|
└ Done. Now run:

cd odp1
npm install
npm run dev
```

Приказане три команде једну по једну унети у терминал. Прва команда служи за промену директоријума у директоријум где је креиран празан шаблон за пројекат, док друга команда служи да покрене инсталацију неопходних пакета помоћу **Node Package Manager**.

Прве две команде се углавном покрећу једном, и то у ситуацији тек креiranог новог пројекта. Последња команда служи како би се **покренуло** локално окружење у коме ће се извршавати апликација, при чему се свака измена у извornом коду одмах одражава и на корисничком интерфејсу.

Када се локално окружење покрене приказаће се веб адреса преко које приступамо корисничком интерфејсу:

- Local: http://localhost:5173/
- Network: use --host to expose
- press h + enter to show help

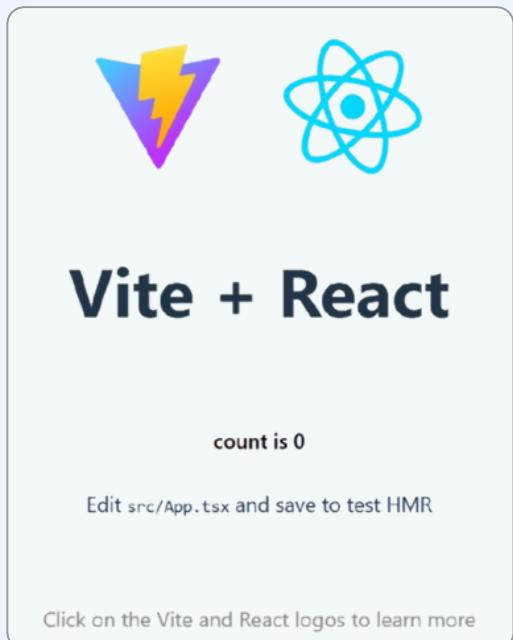


# УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Вежба

Уносом веб адресе у претраживач добићемо почетну страницу веб апликације:



Структура пројекта састоји се од више под-директоријума и изворних датотека:



- ***node\_modules*** садржи све пакете неопходне за рад пројекта
- ***public*** садржи датотеке које су дељени ресурси (слике, документи, и сл.) које се могу било где користити у оквиру веб апликације
- ***src*** садржи изворни код пројекта и у њему се пише комплетан изворни код (класе, компоненте, сервиси...)
- ***main.tsx*** је **TypeScript** датотека из које почиње целокупно извршавање веб апликације
- ***package.json*** садржи списак пакета који се користе у пројекту и на основу њега командом за инсталацију пакета се преузимају сви пакети са ***node package*** репозиторијума
- ***vite.config.ts*** је конфигурациона датотека специфична за **Vite** развојно окружење



# УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Задатак

Потребно је креирати **React** апликацију која садржи форму са два поља:

- **Име и презиме** – текстуално поље у које корисник уноси своје име.
- **Број поена** – нумеричко поље за унос броја поена у опсегу од 0 до 100.

На основу унете вредности поена, апликација треба да израчуна оцену и прикаже поруку у следећем формату:

[Име и презиме] ће имати оцену [оцену] из ОДП-а.

Предложени изглед апликације је:

### Оцена из ОДП-а

Име и презиме:  
Данијел Јовановић

Број поена (0–100):  
87

**Прикажи оцену**

Данијел Јовановић ће имати оцену 9 из ОДП-а.



# УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Решење

У *App.tsx* потребно је урадити следеће:

```
import { useState } from 'react'
import './App.css'

function App() {
  const [ime, setIme] = useState<string>('')
  const [poeni, setPoeni] = useState<number | ''>('')
  const [poruka, setPoruka] = useState<string>('')

  const izracunajOcenu = () => {
    if (ime.trim() === '' || poeni === '') {
      setPoruka('Молимо унесите и име и број поена.')
      return
    }

    const brojPoena = Number(poeni)
    let ocena: string

    if (brojPoena <= 50) ocena = 'недовољан (5)'
    else if (brojPoena <= 60) ocena = '6'
    else if (brojPoena <= 70) ocena = '7'
    else if (brojPoena <= 80) ocena = '8'
    else if (brojPoena <= 90) ocena = '9'
    else ocena = '10'

    setPoruka(`${ime} ће имати оцену ${ocena} из ОДП-а.`)
  }

  return (
    <div className="form-container">
      <h1>Оцена из ОДП-а</h1>

      <div className="input-group">
        <label htmlFor="ime">Име и презиме:</label>
        <input
          id="ime"
          type="text"
          value={ime}
          onChange={(e) => setIme(e.target.value)}
          placeholder="Унесите име и презиме"
        />
      </div>
    </div>
  )
}
```



## УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Решење

У *App.tsx* потребно је урадити следеће:

```
<div className="input-group">
  <label htmlFor="poeni">Број поена (0-100):</label>
  <input
    id="poeni"
    type="number"
    value={poeni}
    onChange={(e) => {
      const value = e.target.value
      if (value === '') {
        setPoeni('')
      } else {
        const broj = Number(value)
        if (broj >= 0 && broj <= 100) {
          setPoeni(broj)
        }
      }
    }}
    placeholder="Унесите број поена"
  />
</div>

<button onClick={izracunajOcenu}>Прикажи оцену</button>

{poruka && <p>{poruka}</p>}
</div>
)
}

export default App
```



# УВОД У REACT. VITE.JS

ВЕЖБЕ 1

## Решење

У *Index.css* потребно је урадити следеће:

```
:root {
    font-family: system-ui, Avenir, Helvetica, Arial, sans-serif;
    line-height: 1.5;
    font-weight: 400;
}

h1 {
    font-size: 2.4em;
    line-height: 1.1;
    margin-bottom: 1rem;
    text-align: center;
}

button {
    border-radius: 8px;
    border: 1px solid transparent;
    padding: 0.6em 1.2em;
    font-size: 1em;
    font-family: inherit;
    width: 100%;
}

.form-container {
    max-width: 400px;
    padding: 2rem;
}

.input-group {
    margin-bottom: 1.5rem;
}

label {
    margin-bottom: 0.5rem;
    font-weight: 500;
    text-align: left;
}

input {
    width: 100%;
    padding: 0.6rem;
    border-radius: 6px;
    border: 1px solid #ccc;
}
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2



## Увод

**Валидација података** представља један од кључних корака у процесу обраде података у веб апликацијама. Њена основна сврха је да осигура да унесени подаци испуњавају очекиване критеријуме пре него што се употребе за даљу **логичку обраду**, **чување у бази података** или **приказ резултата**. Примена валидације омогућава **спречавање логичких грешака, неконсистентности**, нежељених резултата, као и потенцијалних **безбедносних ризика**.

У контексту **React** апликација, валидација често укључује проверу да ли су **обавезна поља** попуњена, да ли унети бројеви припадају дозвољеном **интервалу**, као и да ли текстуалне вредности имају **очекиван формат**. На тај начин се обезбеђује да апликација функционише стабилно и у складу са дефинисаним правилима пословне логике.

**Локално складиште података** (енг. *local storage*) представља механизам за чување података на страни клијента, односно у прегледачу корисника. То је део **Web Storage API**-ја који омогућава апликацијама да трајно чувају податке у виду **парова кључ-вредност**, чак и након што се страница освежи или затвори. За разлику од **сесијског складишта (session storage)**, подаци у **локалном складишту** се **не бришу аутоматски** након затварања картице или прегледача, већ остају **доступни** све док их **апликација** експлицитно не уклони.

Честа примена је чување **аутентификацијоног токена** у локалном складишту, како би се серверска апликација ослободила додатне логике чување токена.

## Потребан софтвер

- **Visual Studio Code**  
<https://code.visualstudio.com/download>
- **Node.js**  
<https://nodejs.org/en/download>

## Питања

- Шта је сврха валидације података у веб апликацијама?
- Зашто је важно валидирати унос пре него што се обради?
- Која је улога ***isNaN()*** функције у валидацији?
- Да ли је препоручљиво вршити валидацију само на клијентској страни?
- Шта је **localStorage** и како се чувају подаци?
- Која је разлика између **localStorage** и **sessionStorage**?
- Како се у **React**-у чита вредност из **localStorage**-а приликом учитавања компоненте?
- Објасни улогу **useEffect** хука у чувању података у **localStorage**-у?



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Методи валидације података

**Клијентска и серверска валидација** представљају два основна приступа провери исправности података у веб апликацијама. **Клијентска валидација** се извршава директно у **прегледачу корисника**, пре него што се подаци пошаљу серверу.

Њена основна улога је да омогући брузу и интерактивну повратну информацију кориснику, чиме се побољшава корисничко искуство и смањује број непотребних захтева ка серверу. Реализује се помоћу **HTML5 атрибута** (*required*, *min*, *max*, *pattern*), **JavaScript** логике или **библиотека** као што су *Formik*, *Yup* и *React Hook Form*. Ипак, иако је ефикасна и практична, **клијентска валидација није довољна** са становишта безбедности, јер се лако може заобићи путем алата за измену кода у прегледачу. Пример **клијентске валидације**:

```
const handleSubmit = () => {
  if (ime.trim() === '') {
    alert('Име је обавезно.')
    return
  }
  if (poeni < 0 || poeni > 100) {
    alert('Поени морају бити у опсегу 0–100.')
    return;
  }
  // Ако је све у реду, шаље се серверу
}
```

**Серверска валидација** подразумева **проверу података** након што они **стигну на сервер**. Она се спроводи у окружењу апликационог сервера, у програмским језицима као што су *Node.js*, *Python* и сл. овај вид валидације је **поузданiji** и **безбедниji** јер није подложен манипулатијама од стране крањег корисника.

**Серверска валидација** је неопходна за критичне провере као што су провера **аутентичности**, **интегритета података**, **права приступа** и **доследности** у бази података. У пракси, најбоље резултате даје **комбиновање обе технике** – клијентска валидација за унапређење интерфејса, а серверска за заштиту система и података.

```
app.post('/rezultat', (req, res) => {
  const { ime, poeni } = req.body

  if (!ime || ime.trim() === '')
    return res.status(400).json({ error: 'Име је обавезно.' })

  if (isNaN(poeni) || poeni < 0 || poeni > 100)
    return res.status(400).json({ error: 'Поени морају бити између 0 и 100.' })

  // Даље обрађивање података
  res.status(200).json({ message: 'Подаци су успешно примљени.' })
})
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Local Storage

Метода за чување вредности у **local storage** прихвата као параметар и **кључ** под којим ће се податак чувати, као и саму **вредност** која се чува као **текстуалну вредност**. Битно је нагласати да се једино текстуалне вредности могу чувати у локалном складишту.

```
function SačuvajVrednostPoKljuču(key: string, value: string): boolean {
    try {
        localStorage.setItem(key, value)
        return true
    } catch (error) {
        console.error(`Грешка при чувању у localStorage за кључ '${key}':`, error)
        return false
    }
}
```

Метода за читање вредности из **local storage** прихвата као параметар **кључ** под којим је податак сачуван, а као повратну вредност враћа вредност у случају да она и постоји под траженим кључем.

```
function PročitajVrednostPoKljuču(key: string): string | null {
    try {
        return localStorage.getItem(key)
    } catch (error) {
        console.error(`Грешка при читању из localStorage за кључ '${key}':`, error)
        return null
    }
}
```

Метода за уклањање вредности из **local storage** прихвата као параметар **кључ** под којим је податак сачуван, а у случају да она постоји под траженим кључем биће обрисана.

```
function ObrišivrednostPoKljuču(key: string): boolean {
    try {
        localStorage.removeItem(key)
        return true
    } catch (error) {
        console.error(`Грешка при брисању из localStorage за кључ '${key}':`, error)
        return false
    }
}
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Задатак

Потребно је креирати **React компоненту** која садржи форму са два поља:

- **Корисничко име** – текстуално поље у које корисник уноси произвољно корисничко име.
- **Лозинка** – текстуално поље у које корисник уноси произвољно лозинку.

Неопходно је урадити **клијентску валидацију података** тако да уноси за оба поља имају **минимално 3 карактера а највише 20 карактера**. За валидацију користити и **HTML атрибуте** и валидацију у оквиру методе за пријем података.

Када корисник унесе податке и испуни све услове валидације, кликом на дугме **Пријава** креира се нови запис у **local storage** под кључем **authToken** и формира се у формату:

[Корисничко име]\_token

Након пријаве кориснику се приказује компонента **Контролна табла** која приказује који корисник је тренутно пријављен, тренутно датум и време као и дугме за **Напуштање контролне табле** чији кликом се из **local storage** уклања вредност под кључем **authToken** и кориснику поново приказује форма за унос података.

Компонента са формом за унос података се приказује у случају да корисник **претходно није унео** податке, односно да се у **local storage** под кључем **authToken** не налази ниједна вредност.

Компонента са приказом информација се приказује у случају да је корисник **претходно унео** податке, односно да се у **local storage** под кључем **authToken** налази вредност у траженом формату.

Предложени изглед је:

## Пријава

Корисничко име:

Унесите корисничко име

Лозинка:

Унесите лозинку

Пријава

## Контролна табла

Пријављени корисник: **danijelj01**

Датум и време: 19. 6. 2025. 11:52:40

Напусти контролну таблу



## ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Решење

У *Index.css* потребно је урадити следеће:

```
:root {  
    font-family: system-ui, Avenir, Helvetica, Arial, sans-serif;  
    line-height: 1.5;  
    font-weight: 400;  
}  
  
h1 {  
    font-size: 2.4em;  
    line-height: 1.1;  
    margin-bottom: 1rem;  
    text-align: center;  
}  
  
button {  
    border-radius: 8px;  
    border: 1px solid transparent;  
    padding: 0.6em 1.2em;  
    font-size: 1em;  
    font-family: inherit;  
    width: 100%;  
}  
  
.form-container {  
    max-width: 400px;  
    padding: 2rem;  
}  
  
.input-group {  
    margin-bottom: 1.5rem;  
}  
  
label {  
    margin-bottom: 0.5rem;  
    font-weight: 500;  
    text-align: left;  
}  
  
input {  
    width: 100%;  
    padding: 0.6rem;  
    border-radius: 6px;  
    border: 1px solid #ccc;  
}
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Решење

У `src → helpers → local_storage.ts` потребно је урадити следеће:

```
function SačuvajVrednostPoKljuču(key: string, value: string): boolean {
    try {
        localStorage.setItem(key, value)
        return true
    } catch (error) {
        console.error(`Грешка при чувању у localStorage за кључ '${key}':`, error)
        return false
    }
}

function PročitajVrednostPoKljuču(key: string): string | null {
    try {
        return localStorage.getItem(key)
    } catch (error) {
        console.error(`Грешка при читању из localStorage за кључ '${key}':`, error)
        return null
    }
}

function ObrišiVrednostPoKljuču(key: string): boolean {
    try {
        localStorage.removeItem(key)
        return true
    } catch (error) {
        console.error(`Грешка при брисању из localStorage за кључ '${key}':`, error)
        return false
    }
}

export { SačuvajVrednostPoKljuču, PročitajVrednostPoKljuču, ObrišiVrednostPoKljuču };
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Решење

У `src → components → kontrolna_tabla → KontrolnaTabla.tsx` потребно је урадити следеће:

```
import { PročitajVrednostPoKljuču, ObrišiVrednostPoKljuču } from "../../helpers/local_storage"

type Props = {
  onLogout: () => void
}

export default function KontrolnaTabla({ onLogout }: Props) {
  const authToken = PročitajVrednostPoKljuču('authToken')
  const korisnickoIme = authToken ? authToken.replace('_token', '') : 'Непознато'
  const trenutnoVreme = new Date().toLocaleString()

  const handleLogout = () => {
    ObrišiVrednostPoKljuču('authToken')
    onLogout()
  }

  return (
    <div className="form-container">
      <h1>Контролна табла</h1>
      <p>Пријављени корисник: <strong>{korisnickoIme}</strong></p>
      <p>Датум и време: {trenutnoVreme}</p>
      <button onClick={handleLogout}>Напусти контролну таблу</button>
    </div>
  )
}
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Решење

У `src → components → prijava → FormaZaPrijavu.tsx` потребно је урадити следеће:

```
import { useState } from 'react'
import { SačuvajVrednostPoKljuču } from '../.../helpers/local_storage'

type Props = {
  onLoginSuccess: () => void
}

export default function FormaZaPrijavu({ onLoginSuccess }: Props) {
  const [korisnickoIme, setKorisnickoIme] = useState<string>('')
  const [lozinka, setLozinka] = useState<string>('')

  const podnesiFormu = (e: React.FormEvent) => {
    e.preventDefault()

    if (korisnickoIme.length < 3 || korisnickoIme.length > 20 || lozinka.length < 3 || lozinka.length > 20) {
      alert('Корисничко име и лозинка морају имати између 3 и 20 карактера.')
      return;
    }

    const token = `${korisnickoIme}_token`
    const uspeh = SačuvajVrednostPoKljuču('authToken', token)
    if (uspeh) onLoginSuccess()
  }

  return (
    <div className="form-container">
      <h1>Пријава</h1>
      <form onSubmit={podnesiFormu}>
        <div className="input-group">
          <label htmlFor="username">Корисничко име:</label>
          <input
            id="username"
            type="text"
            value={korisnickoIme}
            onChange={(e) => setKorisnickoIme(e.target.value)}
            placeholder="Унесите корисничко име"
            minLength={3}
            maxLength={20}
            required
          />
        </div>
      </form>
    </div>
  )
}
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Решење

У `src → components → prijava → FormaZaPrijavu.tsx` потребно је урадити следеће:

```
<div className="input-group">
  <label htmlFor="password">Лозинка:</label>
  <input
    id="password"
    type="password"
    value={lozinka}
    onChange={(e) => setLozinka(e.target.value)}
    placeholder="Унесите лозинку"
    minLength={3}
    maxLength={20}
    required
  />
</div>

<button type="submit">Пријава</button>
</form>
</div>
)
```



# ВАЛИДАЦИЈА ПОДАТАКА. LOCAL STORAGE

ВЕЖБЕ 2

## Решење

У `src → App.tsx` потребно је урадити следеће:

```
import { useState, useEffect } from "react";
import { PročitajVrednostPoKljuču } from "./helpers/local_storage";
import "./index.css";
import KontrolnaTabla from "./components/kontrolna_tabla/KontrolnaTabla";
import FormaZaPrijavu from "./components/prijava/FormaZaPrijavu";

function App() {
  const [prijavljen, setPrijavljen] = useState<boolean>(false)

  useEffect(() => {
    const token = PročitajVrednostPoKljuču('authToken')
    if (token) {
      setPrijavljen(true)
    }
  }, [])

  return prijavljen ? (
    <KontrolnaTabla onLogout={() => setPrijavljen(false)} />
  ) : (
    <FormaZaPrijavu onLoginSuccess={() => setPrijavljen(true)} />
  )
}

export default App;
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Увод

**Клијент-сервер** архитектура представља један од најзаступљенијих модела у дизајну модерних веб апликација. Њена основна сврха је јасна **дистрибуција одговорности** између два ентитета: **клијента**, који је задужен за **интеракцију са корисником** и презентацију података, и **сервера**, који **обрађује** захтеве, **управља** логиком и **чува** податке. Оваква подела омогућава лакше одржавање, скалабилност и јасно дефинисан ток комуникације путем **HTTP(S)** протокола.

Применом **SOLID принципа** и концепата чисте архитектуре у оквиру **клијент-сервер** модела, омогућава се развој софтвера који је лак за тестирање, проширив, и отпоран на промене. **SOLID** принципи помажу у дефинисању класа и модула који имају јасну сврху, **минималне међузависности** и модуларан дизајн, док чиста архитектура издваја доменску логику од инфраструктурних детаља, чиме се омогућава независност од радног оквира (**framework**), базе података или корисничког интерфејса.

У контексту веб апликација, примена ових принципа доводи до тога да клијентска и серверска страна буду логички одвојене, али истовремено међусобно усклађене у погледу интерфејса и протокола комуникације. То омогућава паралелни развој обе апликације, уз минималан ризик од нарушавања интегритета апликације.

## Потребан софтвер

- **Visual Studio Code**  
<https://code.visualstudio.com/download>
- **MySQL Community Server 8.4.5**  
<https://dev.mysql.com/get/mysql-8.4.5-winx64.msi>
- **Node.js**  
<https://nodejs.org/en/download>
- **MySQL Workbench**  
<https://dev.mysql.com/workbench-8.0.42-winx64.msi>

## Питања

- Шта подразумевамо под појмом **клијент-сервер** архитектуре и које су њене кључне карактеристике?
- Које су главне предности и мане **клијент-сервер** модела у односу на **монолитне** апликације?
- Како се обично одвија комуникација између клијента и сервера у веб апликацијама?
- Који су најчешћи формати за размену података између клијента и сервера?
- Шта представља акроним **SOLID** и зашто су ови принципи важни у програмирању?
- Како се примена **Single Responsibility Principle (SRP)** одражава на архитектуру веб сервиса?
- Шта подразумева појам **чиста архитектура** и који су њени основни концепти?
- Које слојеве обично садржи **чиста архитектура** и која је њихова улога?
- Које су предности коришћења **Dependency Injection**-а у оквиру чисте архитектуре?
- Како комбиновати принципе **SOLID** и **чисту архитектуру** у реалним пројектима?



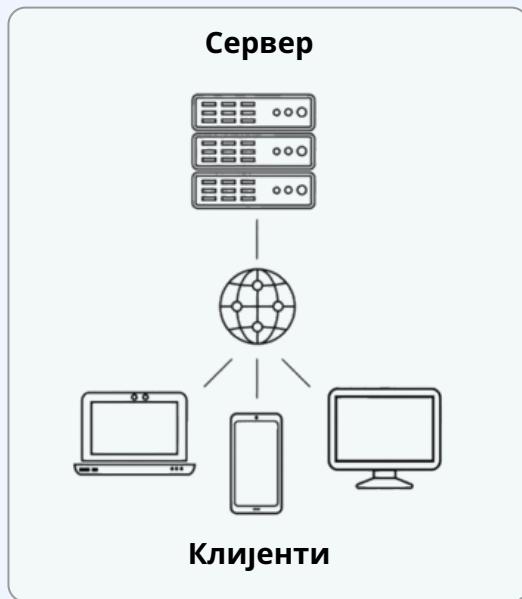
# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3



## Клијент-сервер архитектура

**Клијент-сервер** архитектура се заснива на идеји независне, али усклађене комуникације између два ентитета, који међусобно комуницирају путем стандардизованих интерфејса, као што су **HTTP** протокол и формати попут **JSON**-а. Овим приступом се постиже боља изолованост слојева, што је у складу са савременим софтверским принципима као што су **Separation of Concerns (SoC)** и **Single Responsibility Principle (SRP)**.



У случајевима када иста особа развија и **клијентску** и **серверску** страну апликације, примена овакве архитектуре доноси вишеструке предности.

Пре свега, омогућава се пуна контрола над читавим током података, од уноса у корисничком интерфејсу, преко обраде на серверу, до приказа резултата.

Ово значајно поједностављује процес интеграције и откривања грешака, јер се сва логика налази унутар јединственог контекста који је програмеру познат.

Оваква архитектура омогућава паралелни развој клијентског и серверског дела, уз примену **Mock** интерфејса или тестних података.

То је посебно корисно у фазама развоја када **серверски API** још није потпуно завршен.

На **серверској** страни, логика се може организовати у слојеве попут **контролера**, **сервиса** и **репозиторијума**, чиме се обезбеђује висока кохезија и ниска спретност компоненти.

На **клијентској** страни, у оквиру **React** апликације, могуће је примењивати сличан структурни приступ кроз **компоненте**, **hook-ове**, **сервисе** и **контекст**.

Пројекат за **клијентску апликацију** креира се на исти начин (**вежбе 1**), уз то да је потребно и инсталацији библиотеку **axios** која омогућава слање и пријем података са **серверске апликације**.

<ul style="list-style-type: none"> <li>✓ client</li> <li>  ✓ api_services</li> <li>    ✓ auth_api</li> <li>      <b>TS</b> AuthApi.ts</li> <li>      <b>TS</b> IAuthApi.ts</li> <li>    &gt; user_api</li> <li>    &gt; components</li> <li>    ✓ contexts</li> <li>      &gt; app_context</li> <li>      ✓ auth_context</li> <li>      &gt; helpers</li> <li>      &gt; hooks</li> <li>      &gt; models</li> <li>      &gt; pages</li> <li>      <b>App.tsx</b></li> <li>      <b># index.css</b></li> <li>      <b>TS main.ts</b></li> </ul>	<ul style="list-style-type: none"> <li>✓ server</li> <li>  ✓ Database</li> <li>    ✓ connection</li> <li>    ✓ repositories</li> <li>  ✓ Domain</li> <li>    &gt; constants</li> <li>    &gt; DTOs</li> <li>    &gt; enums</li> <li>    &gt; models</li> <li>    &gt; repositories</li> <li>    &gt; services</li> <li>    &gt; types</li> <li>    &gt; Services</li> <li>  ✓ WebAPI</li> <li>    &gt; controllers</li> <li>    &gt; middleware</li> <li>    <b>TS index.ts</b></li> </ul>
---	--

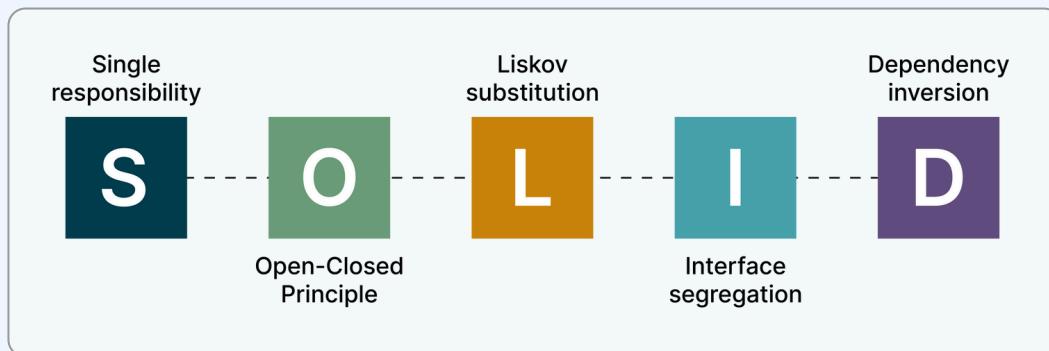


# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## SOLID принципи

SOLID представља скуп од пет принципа који имају за циљ унапређење структуре, одрживости и проширивости софтверских система. Принципи су формулисани с намером да помогну у стварању система који су отпорни на промене, једноставни за разумевање и лако тестирујући.



### ***Single Responsibility Principle***

Свака класа или модул треба да има само једну одговорност, односно да буде задужена за један аспект функционалности.

### ***Open/Closed Principle***

Софтверски ентитети треба да буду отворени за проширење, али затворени за измене. То омогућава додавање нових функционалности без нарушавања постојећег кода.

### ***Liskov Substitution Principle***

Наслеђени типови морају бити у стању да замене своје родитеље без нарушавања функционалности. Овај принцип осигурује да се хијерархије класа користе на предвидљив и логички исправан начин.

### ***Interface Segregation Principle***

Корисници интерфејса не би требало да буду приморани да имплементирају методе које не користе. Уместо великих и општих интерфејса, боље је креирати више мањих, специфичних интерфејса, што поједностављује имплементацију и повећава флексибилност система.

### ***Dependency Inversion Principle***

Модули вишег нивоа не би требало да зависе од модула нижег нивоа, већ обоји треба да зависе од апстракција. Применом овог принципа, могуће је лако заменити конкретне имплементације – као што је различит тип базе података или спољашњи API – без утицаја на основну пословну логику.



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Задатак

Потребно је креирати **клијентску апликацију** која садржи форму са два поља:

- **Корисничко име** – текстуално поље у које корисник уноси корисничко име,
- **Лозинка** – текстуално поље у које корисник уноси лозинку.

Неопходно је урадити и **клијентску и серверску валидацију података** тако да уноси за оба поља имају **минимално 3 карактера а највише 20 карактера**. За валидацију користити и **HTML атрибуте** и валидацију у оквиру методе за пријем података.

Када корисник унесе податке и испуни све услове валидације, кликом на дугме **Пријава серверској апликацији** се шаље захтев са унетим подацима, **серверска апликација** захтев приhvата и проверава у **бази података** (неопходно је креирати табелу, **лозинке** у бази података се морају **хеширати**) да ли корисник постоји:

- ако корисник **постоји** и унео је **тачну лозинку** сервер у одговору **клијентској апликацији** на захтев враћа: **ID корисника и корисничко име**,
- ако корисник **постоји** и унео је **нетачне податке** сервер у одговору **клијентској апликацији** на захтев враћа **грешку** да нису унети исправни подаци,
- ако корисник **не постоји**, потребно је урадити **регистрацију** корисника путем форме и послати податке серверу кликом на дугме **Регистрација**, ако корисник не постоји сервер креира новог корисника, и у одговору на захтев враћа: **ID корисника и корисничко име**.

**Клијентска апликација** приhvата одговор **сервера** и ако је пријава или регистрација **успешна** сачуваће **податке** у **local storage** и приказати их у оквиру компоненте **Контролна табла**.

У случају да пријава или регистрација **није успешна**, приказаће **грешку** у оквиру **форме за аутентификацију** и поља за унос података вратити у иницијално (непопуњено) стање.

### Пријава

Корисничко име:

Лозинка:

**Пријава**

Немате налог? Региструјте се

### Контролна табла

ID: 3

Корисничко име: danijelj011

Датум и време: 22. 6. 2025. 20:29:02

Напусти контролну таблу



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3



## Решење: Клијент

У *Index.css* потребно је урадити следеће:

```
:root {
    font-family: system-ui, Avenir, Helvetica, Arial, sans-serif;
    line-height: 1.5;
    font-weight: 400;
}

h1 {
    font-size: 2.4em;
    line-height: 1.1;
    margin-bottom: 1rem;
    text-align: center;
}

button {
    border-radius: 8px;
    border: 1px solid transparent;
    padding: 0.6em 1.2em;
    font-size: 1em;
    font-family: inherit;
    width: 100%;
}

.form-container {
    max-width: 400px;
    padding: 2rem;
}

.input-group {
    margin-bottom: 1.5rem;
}

label {
    margin-bottom: 0.5rem;
    font-weight: 500;
    text-align: left;
}

input {
    width: 95%;
    padding: 0.6rem;
    border-radius: 6px;
    border: 1px solid #ccc;
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Клијент

У `src → components → kontrolna_tabla → KontrolnaTabla.tsx` потребно је урадити следеће:

```
import { PročitajVrednostPoKljuču, ObrišiVrednostPoKljuču } from "../../helpers/local_storage"

type KontrolnaTablaProps = {
  onLogout: () => void
}

export default function KontrolnaTabla({ onLogout }: KontrolnaTablaProps) {
  const token = PročitajVrednostPoKljuču("authToken")

  if (!token || !token.includes("/")) {
    onLogout();
    return;
  }

  const [korisnickoIme, id] = token.split("/")

  const handleLogout = () => {
    ObrišiVrednostPoKljuču("authToken");
    onLogout();
  }

  return (
    <div className="form-container">
      <h1>Контролна таблица</h1>
      <p><strong>ID:</strong> {id}</p>
      <p><strong>Корисничко име:</strong> {korisnickoIme}</p>
      <p><strong>Датум и време:</strong> {new Date().toLocaleString()}</p>
      <button onClick={handleLogout}>Напусти контролну таблицу</button>
    </div>
  )
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Клијент

У `src → components → autentifikacija → AutentifikacionaForma.tsx` потребно је урадити следеће:

```
import { useState } from "react";
import { validacijaPodatakaAuth } from "../../api_services/validators/auth/AuthValidator";
import { SačuvajVrednostPoKljuču } from "../../helpers/local_storage";
import type { IAuthAPIService } from "../../api_services/auth/IAuthAPIService";

type AuthFormProps = {
  authApi: IAuthAPIService;
  onLoginSuccess: () => void;
};

export default function AutentifikacionaForma({
  authApi,
  onLoginSuccess,
}: AuthFormProps) {
  const [korisnickoIme, setKorisnickoIme] = useState("");
  const [lozinka, setLozinka] = useState("");
  const [greska, setGreska] = useState("");
  const [jeRegistracija, setJeRegistracija] = useState(false);

  const podnesiFormu = async (e: React.FormEvent) => {
    e.preventDefault();

    const validacija = validacijaPodatakaAuth(korisnickoIme, lozinka);
    if (!validacija.uspesno) {
      setGreska(validacija.poruka ?? "Неисправни подаци");
      return;
    }

    const odgovor = jeRegistracija
      ? await authApi.registracija(korisnickoIme, lozinka)
      : await authApi.prijava(korisnickoIme, lozinka);

    if (odgovor.success && odgovor.data) {
      const token = `${odgovor.data.korisnickoIme}/${odgovor.data.id}`;
      SačuvajVrednostPoKljuču("authToken", token);
      onLoginSuccess();
    } else {
      setGreska(odgovor.message);
      setKorisnickoIme("");
      setLozinka("");
    }
  };
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Клијент

У `src → components → autentifikacija → AutentifikacionaForma.tsx` потребно је урадити следеће:

```

return (
  <div className="form-container">
    <h1>{jeRegistracija ? "Регистрација" : "Пријава"}</h1>
    <form onSubmit={podnesiFormu}>
      <div className="input-group">
        <label htmlFor="username">Корисничко име:</label>
        <input
          id="username"
          type="text"
          value={korisnickoIme}
          onChange={(e) => setKorisnickoIme(e.target.value)}
          placeholder="Унесите корисничко име"
          minLength={3}
          maxLength={20}
          required
        />
      </div>

      <div className="input-group">
        <label htmlFor="password">Лозинка:</label>
        <input
          id="password"
          type="password"
          value={lozinka}
          onChange={(e) => setLozinka(e.target.value)}
          placeholder="Унесите лозинку"
          minLength={6}
          maxLength={20}
          required
        />
      </div>

      {greska && <p style={{ color: "red" }}>{greska}</p>}

      <button type="submit">{jeRegistracija ? "Регистрација" : "Пријава"}</button>
    </form>
    <button onClick={() => setJeRegistracija(!jeRegistracija)}>
      {jeRegistracija ? "Имате налог? Пријавите се" : "Немате налог? Региструјте се"}
    </button>
  </div>
);
}

```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Клијент

У `src → helpers → local_storage.ts` потребно је урадити следеће:

```
function SačuvajVrednostPoKljuču(key: string, value: string): boolean {
    try {
        localStorage.setItem(key, value)
        return true
    } catch (error) {
        console.error(`Грешка при чувању у localStorage за кључ '${key}'`, error)
        return false
    }
}

function PročitajVrednostPoKljuču(key: string): string | null {
    try {
        return localStorage.getItem(key)
    } catch (error) {
        console.error(`Грешка при читању из localStorage за кључ '${key}'`, error)
        return null
    }
}

function ObrišiVrednostPoKljuču(key: string): boolean {
    try {
        localStorage.removeItem(key)
        return true
    } catch (error) {
        console.error(`Грешка при брисању из localStorage за кључ '${key}'`, error)
        return false
    }
}

export { SačuvajVrednostPoKljuču, PročitajVrednostPoKljuču, ObrišiVrednostPoKljuču };
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Клијент

У `src → models → auth → UserLoginDto.ts` потребно је урадити следеће:

```
export interface UserLoginDto {
    id: number;
    korisnickoIme: string;
}
```

У `src → types → auth → AuthResponse.ts` потребно је урадити следеће:

```
import type { UserLoginDto } from "../../models/auth/UserLoginDto";

export interface AuthResponse {
    success: boolean;
    message: string;
    data?: UserLoginDto;
}
```

У `src → api_services → validators → AuthValidator.ts` потребно је урадити следеће:

```
import type { RezultatValidacije } from "../../types/validation/validationResult";

export function validacijaPodatakaAuth(korisnickoIme?: string, lozinka?: string): RezultatValidacije {
    if (!korisnickoIme || !lozinka) {
        return { uspesno: false, poruka: 'Korisničko ime i lozinka su obavezni.' };
    }

    if (korisnickoIme.length < 3) {
        return { uspesno: false, poruka: 'Korisničko ime mora imati najmanje 3 karaktera.' };
    }

    if (lozinka.length < 6) {
        return { uspesno: false, poruka: 'Lozinka mora imati najmanje 6 karaktera.' };
    }

    if (lozinka.length > 20) {
        return { uspesno: false, poruka: 'Lozinka može imati najviše 20 karaktera.' };
    }

    return { uspesno: true };
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Клијент

У `src → api_services → auth → IAuthAPIService.ts` потребно је урадити следеће:

```
import type { AuthResponse } from "../../types/auth/AuthResponse";

export interface IAuthAPIService {
    prijava(korisnickoIme: string, lozinka: string): Promise<AuthResponse>;
    registracija(korisnickoIme: string, lozinka: string): Promise<AuthResponse>;
}
```

У `src → api_services → auth → AuthAPIService.ts` потребно је урадити следеће:

```
import type { AuthResponse } from "../../types/auth/AuthResponse";
import type { IAuthAPIService } from "./IAuthAPIService";
import axios from "axios";

const API_URL: string = import.meta.env.VITE_API_URL + "auth";

export const authApi: IAuthAPIService = {
    async prijava(korisnickoIme: string, lozinka: string): Promise<AuthResponse> {
        try {
            const res = await axios.post<AuthResponse>(`#${API_URL}/login`,
                { korisnickoIme, lozinka });

            return res.data;
        } catch (error) {
            let message = "Greška prilikom prijave.";
            if (axios.isAxiosError(error)) message = error.response?.data?.message || message;
            return { success: false, message, data: undefined };
        }
    },
    async registracija(korisnickoIme: string, lozinka: string): Promise<AuthResponse> {
        try {
            const res = await axios.post<AuthResponse>(`#${API_URL}/register`,
                { korisnickoIme, lozinka });

            return res.data;
        } catch (error) {
            let message = "Greška prilikom registracije.";
            if (axios.isAxiosError(error)) message = error.response?.data?.message || message;
            return { success: false, message, data: undefined };
        }
    },
};
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Клијент

У `src → App.tsx` потребно је урадити следеће:

```
import { useState, useEffect } from 'react'
import { PročitajVrednostPoKljuču } from './helpers/local_storage'
import KontrolnaTabla from './components/kontrolna_tabla/KontrolnaTabla'
import AutentifikacionaForma from './components/autentifikacija/AutentifikacionaForma'
import { authApi } from './api_services/auth/AuthAPIService';

function App() {
  const [prijavljen, setPrijavljen] = useState<boolean>(false)

  useEffect(() => {
    const token = PročitajVrednostPoKljuču('authToken')
    if (token) {
      setPrijavljen(true)
    }
  }, [])

  return prijavljen ? (
    <KontrolnaTabla onLogout={() => setPrijavljen(false)} />
  ) : (
    <AutentifikacionaForma authApi={authApi} onLoginSuccess={() => setPrijavljen(true)} />
  )
}

export default App;
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3



## Решење: Клијент

У **.env** потребно је урадити следеће:

```
VITE_API_URL=http://localhost:4000/api/v1/
```

## Решење: База података

У **MySQL Workbench** потребно је креирати базу података и креирати табелу:

```
-- Kreiranje baze podataka
CREATE DATABASE IF NOT EXISTS DEFAULT_DB;

-- Koriscenje default baze podataka
USE DEFAULT_DB;

-- Kreiranje tabele za korisnike
CREATE TABLE IF NOT EXISTS users (
    id INT PRIMARY KEY AUTO_INCREMENT,
    korisnickoIme VARCHAR(50) UNIQUE,
    lozinka VARCHAR(500)
);
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У `src → Domain → repositories → users → IUserRepository.ts` потребно је урадити следеће:

```
import { User } from "../../models/User";

export interface IUserRepository {
    create(user: User): Promise<User>;
    getById(id: number): Promise<User>;
    getByUsername(korisnickoIme: string): Promise<User>;
    getAll(): Promise<User[]>;
    update(user: User): Promise<User>;
    delete(id: number): Promise<boolean>;
    exists(id: number): Promise<boolean>;
}
```

У `src → Database → repositories → users → UserRepository.ts` потребно је урадити следеће:

```
import { IUserRepository } from "../../Domain/repositories/users/IUserRepository";
import { User } from "../../Domain/models/User";
import { RowDataPacket, ResultSetHeader } from "mysql2";
import db from "../../connection/DbConnectionPool";

export class UserRepository implements IUserRepository {
    async create(user: User): Promise<User> {
        try {
            const query = `INSERT INTO users (korisnickoIme, lozinka) VALUES (?, ?)`;

            const [result] = await db.execute<ResultSetHeader>(query, [user.korisnickoIme, user.lozinka]);

            if (result.insertId)
                return new User(result.insertId, user.korisnickoIme, user.lozinka);
            return new User();
        } catch { return new User(); }
    }
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У `src → Database → repositories → users → UserRepository.ts` потребно је урадити следеће:

```

async getById(id: number): Promise<User> {
  try {
    const query = `SELECT id, korisnickoIme, lozinka FROM users WHERE id = ?`;
    const [rows] = await db.execute<RowDataPacket[]>(query, [id]);

    if (rows.length > 0) {
      const row = rows[0];
      return new User(row.id, row.korisnickoIme, row.lozinka);
    }
    return new User();
  } catch { return new User(); }
}

async getByUsername(korisnickoIme: string): Promise<User> {
  try {
    const query = `SELECT id, korisnickoIme, lozinka FROM users WHERE korisnickoIme = ?`;
    const [rows] = await db.execute<RowDataPacket[]>(query, [korisnickoIme]);

    if (rows.length > 0) {
      const row = rows[0];
      return new User(row.id, row.korisnickoIme, row.lozinka);
    }

    return new User();
  } catch { return new User(); }
}

async getAll(): Promise<User[]> {
  try {
    const query = `SELECT id, korisnickoIme, lozinka FROM users ORDER BY id ASC`;
    const [rows] = await db.execute<RowDataPacket[]>(query);
    return rows.map(
      (row) => new User(row.id, row.korisnickoIme, row.lozinka)
    );
  } catch { return []; }
}

```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У `src → Database → repositories → users → UserRepository.ts` потребно је урадити следеће:

```
async update(user: User): Promise<User> {
  try {
    const query = `UPDATE users SET korisnickoIme = ?, lozinka = ? WHERE id = ?`;
    const [result] = await db.execute<ResultSetHeader>(query, [
      user.korisnickoIme, user.lozinka, user.id,]);
    if (result.affectedRows > 0)
      return user;
    return new User();
  } catch { return new User(); }
}

async delete(id: number): Promise<boolean> {
  try {
    const query = `DELETE FROM users WHERE id = ?`;
    const [result] = await db.execute<ResultSetHeader>(query, [id]);
    return result.affectedRows > 0;
  } catch { return false; }
}

async exists(id: number): Promise<boolean> {
  try {
    const query = `SELECT COUNT(*) as count FROM users WHERE id = ?`;
    const [rows] = await db.execute<RowDataPacket[]>(query, [id]);
    return rows[0].count > 0;
  } catch { return false; }
}
```

## Решење: Сервер

У `src → Domain → DTOs → auth → UserLoginDto.ts` потребно је урадити следеће:

```
export class UserLoginDto {
  public constructor(
    public id: number = 0,
    public korisnickoIme: string = ''
  ) {}
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У *src → Database → connection → DbConnectionPool.ts* потребно је урадити следеће:

```
import mysql, { Pool } from "mysql2/promise";
import dotenv from "dotenv";

dotenv.config();

const dbConfig = {
  host: process.env.DB_HOST as string,
  user: process.env.DB_USER as string,
  password: process.env.DB_PASSWORD as string,
  database: process.env.DB_NAME as string,
  port: Number(process.env.DB_PORT),
  ssl: {
    rejectUnauthorized: false,
  },
};

const db: Pool = mysql.createPool(dbConfig);

export default db;
```

## Решење: Сервер

У *src → Domain → models → User.ts* потребно је урадити следеће:

```
export class User {
  public constructor(
    public id: number = 0,
    public korisnickoIme: string = '',
    public lozinka: string = ''
  ) {}
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У *src → Domain → services → IAuthService.ts* потребно је урадити следеће:

```
import { UserLoginDto } from "../../DTOs/auth/UserLoginDto";

export interface IAuthService {
    prijava(korisnickoIme: string, lozinka: string): Promise<UserLoginDto>;
    registracija(korisnickoIme: string, lozinka: string): Promise<UserLoginDto>;
}
```

## Решење: Сервер

У *src → Services → auth → AuthService.ts* потребно је урадити следеће:

```
import { UserLoginDto } from "../../Domain/DTOs/auth/UserLoginDto";
import { User } from "../../Domain/models/user";
import { IUserRepository } from "../../Domain/repositories/users/IUserRepository";
import { IAuthService } from "../../Domain/services/auth/IAuthService";
import bcrypt from "bcryptjs";

export class AuthService implements IAuthService {
    private readonly saltRounds: number = parseInt(process.env.SALT_ROUNDS || "10", 10);
    public constructor(private userRepository: IUserRepository) {}

    async prijava(korisnickoIme: string, lozinka: string): Promise<UserLoginDto> {
        const user = await this.userRepository.getByUsername(korisnickoIme);
        if (user.id !== 0 && await bcrypt.compare(lozinka, user.lozinka))
            return new UserLoginDto(user.id, user.korisnickoIme);
        else
            return new UserLoginDto();
    }

    async registracija(korisnickoIme: string, lozinka: string): Promise<UserLoginDto> {
        const existingUser = await this.userRepository.getByUsername(korisnickoIme);
        if (existingUser.id !== 0) return new UserLoginDto();
        const hashedPassword = await bcrypt.hash(lozinka, this.saltRounds);
        const newUser = await this.userRepository.create(new User(0, korisnickoIme,
        hashedPassword));
        if (newUser.id !== 0)
            return new UserLoginDto(newUser.id, newUser.korisnickoIme);
        else
            return new UserLoginDto();
    }
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У `src → WebAPI → Controllers → AuthController.ts` потребно је урадити следеће:

```
import { Request, Response, Router } from 'express';
import { IAuthService } from '../Domain/services/auth/IAuthService';
import { validacijaPodatakaAuth } from '../validators/auth/RegisterValidator';

export class AuthController {
    private router: Router;
    private authService: IAuthService;

    constructor(authService: IAuthService) {
        this.router = Router();
        this.authService = authService;
        this.initializeRoutes();
    }

    private initializeRoutes(): void {
        this.router.post('/auth/login', this.prijava.bind(this));
        this.router.post('/auth/register', this.registracija.bind(this));
    }

    private async prijava(req: Request, res: Response): Promise<void> {
        try {
            const { korisnickoIme, lozinka } = req.body;
            const rezultat = validacijaPodatakaAuth(korisnickoIme, lozinka);
            if (!rezultat.uspesno) {
                res.status(400).json({ success: false, message: rezultat.poruka });
                return;
            }

            const result = await this.authService.prijava(korisnickoIme, lozinka);
            if (result.id !== 0) {
                res.status(200).json({ success: true, message: 'Uspešna prijava', data: result });
                return;
            } else {
                res.status(401).json({ success: false, message: 'Неисправно корисничко име или лозинка' });
                return;
            }
        } catch (error) {
            res.status(500).json({ success: false, message: error });
        }
    }
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У *src → WebAPI → Controllers → AuthController.ts* потребно је урадити следеће:

```
private async registracija(req: Request, res: Response): Promise<void> {
    try {
        const { korisnickoIme, lozinka } = req.body;
        const rezultat = validacijaPodatakaAuth(korisnickoIme, lozinka);
        if (!rezultat.uspesno) {
            res.status(400).json({ success: false, message: rezultat.poruka });
            return;
        }

        const result = await this.authService.registracija(korisnickoIme, lozinka);
        if (result.id !== 0) {
            res.status(201).json({success: true, message: 'Uspešna registracija', data: result});
        } else {
            res.status(401).json({success: false, message: 'Регистрација није успела.'});
        }
    } catch (error) {
        res.status(500).json({success: false, message: error});
    }
}

public getRouter(): Router {
    return this.router;
}
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У `src → app.ts` потребно је урадити следеће:

```
import express from 'express';
import cors from 'cors';
import { IAuthService } from './Domain/services/auth/IAuthService';
import { AuthService } from './Services/auth/AuthService';
import { IUserRepository } from './Domain/repositories/users/IUserRepository';
import { UserRepository } from './Database/repositories/users/UserRepository';
import { AuthController } from './webAPI/controllers/AuthController';

require('dotenv').config();

const app = express();

app.use(cors());
app.use(express.json());

// Repositories
const userRepository: IUserRepository = new UserRepository();

// Services
const authService: IAuthService = new AuthService(userRepository);

// WebAPI routes
const authController = new AuthController(authService);

// Registering routes
app.use('/api/v1', authController.getRouter());

export default app;
```

## Решење: Сервер

У `src → index.ts` потребно је урадити следеће:

```
import app from './app';

const port = process.env.PORT || 5000;

app.listen(port, () => {
  console.log(`Listening: http://localhost:${port}`);
});
```



# КЛИЈЕНТ-СЕРВЕР АРХИТЕКТУРА. SOLID

ВЕЖБЕ 3

## Решење: Сервер

У `.env` потребно је урадити следеће:

```
# DATABASE CONFIGURATION
DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASSWORD=1234
DB_NAME=DEFAULT_DB
DB_SSL_MODE=REQUIRED

# EXPRESS PORT CONFIGURATION
PORT=4000

# JWT TOKEN
JWT_SECRET=OVOJEODPKOVOLIVOLECEKONEZAVOLECEBUDITEUBEDJENIUTODALJENEMAODTOGA

# SALT FOR PASSWORD
SALT_ROUNDS=10
```



# РЕПЛИКАЦИЈА И ОТПОРНОСТ НА ОТКАЗЕ

ВЕЖБЕ 4

## Увод

У модерним веб и *cloud* апликацијама, **поузданост** и **доступност** система су од пресудне важности. Са растом броја корисника и све већим ослањањем на дигиталне услуге, непланирани прекиди рада могу имати озбиљне последице – од губитка прихода до губитка тржнишног удела.

Да би се системи учинили отпорнијим на грешке, користе се различите технике, међу којима су **репликација** и **отпорност на отказе** најзначајније. Ове технике омогућавају системима да наставе са радом и у случају кврова појединих компоненти, али и да убрзају рад захваљујући **дистрибуцији оптерећења**.

Два основна концепта која омогућавају изградњу оваквих система су **репликација** и **отпорност на отказе**. **Репликација** се односи на креирање више копија података или сервиса, чиме се постиже виша доступност и поузданост.

С друге стране, **отпорност на отказе** подразумева способност система да настави са радом упркос отказу појединих компоненти, захваљујући унапређеном дизајну и стратегијама опоравка. Комбинација ова два приступа чини темељ високо-доступних архитектура.

## Потребан софтвер

- **Visual Studio Code**  
<https://code.visualstudio.com/download>
  
- **Node.js**  
<https://nodejs.org/en/download>

## Питања

- Шта је **репликација** у контексту рачунарских система?
- Која је разлика између **синхроне** и **асинхроне репликације**?
- Како се постиже **конзистентност** међу више копија у репликационом систему?
- Шта је **географска репликација** и која је њена намена?
- Како репликација утиче на **скалабилност система**?
- Шта је **отпорност на отказе** (*fault tolerance*)?
- Која је разлика између **fault tolerance** и **high availability**?
- Шта значи **failover** и како функционише?
- Шта је **redundancy** и како се користи у дизајну поузданых система?
- Који су најчешћи извори отказа у дистрибуираним системима?
- Какву улогу има **monitoring** у спречавању потпуног отказа система?



# РЕПЛИКАЦИЈА И ОТПОРНОСТ НА ОТКАЗЕ

ВЕЖБЕ 4

## Репликација

**Репликација** представља технику у којој се подаци или системске компоненте **умножавају** и одржавају на **више физичких** или **логичких локација**. Основни циљ репликације је побољшање **доступности** и **перформанси**, као и смањење ризика од губитка података у случају отказа једне од инстанци.

Постоје различите стратегије репликације, укључујући **синхрону** и **асинхрону репликацију**. Код **синхроне** репликације, све **копије** се **ажурирају истовремено**, чиме се обезбеђује **доследност**, али по цену **латенције**. **Асинхронна репликација**, с друге стране, омогућава **брже одговоре**, али постоји **rizик од губитка података** уколико дође до отказа пре синхронизације.

**Репликација** се примењује у базама података, системима за складиштење датотека, *DNS* инфраструктури и *cloud* платформама. У дистрибуираним системима, често се користи географска репликација, где се подаци умножавају на више континената ради смањења кашњења и обезбеђивања непрекидне доступности.

## Отпорност на отказе

**Отпорност на отказе** (енг. *fault tolerance*) је својство система да **настави са радом** и одржи функционалност упркос **отказу једне** или **више** компоненти. Ова особина се постиже кроз примену различитих дизајнерских образца као што су **редундантност**, **автоматски опоравак**, **детекција грешака** и динамичко преусмеравање саобраћаја.

**Редундантност** подразумева постојање вишке ресурса који могу преузети улогу отказале компоненте. То могу бити резервни сервери, копије података или вишеструки комуникациони канали. Када се открије отказ, систем може извршити тзв. **failover**, односно пребацити рад на функционалну компоненту без прекида у сервису.

У модерним дистрибуираним системима, отпорност на отказе обухвата и концепте као што су **self-healing архитектуре**, у којима системи могу самостално открити и опоравити се од грешака. Отпорност није само техничко питање, већ и стратешко – утиче на углед, пословање и задовољство корисника.



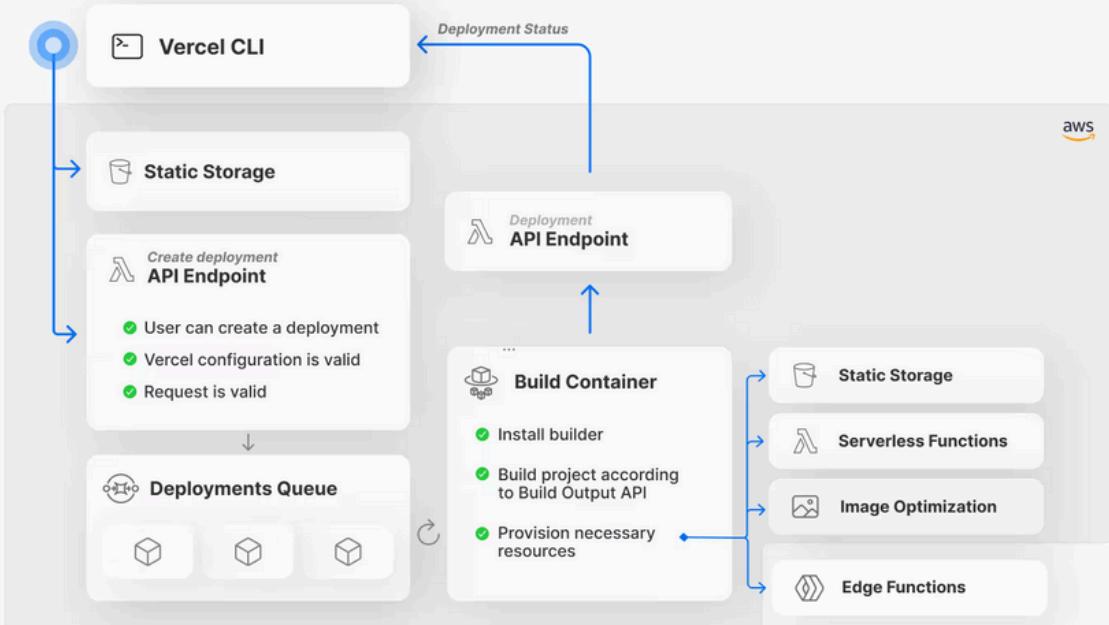
# РЕПЛИКАЦИЈА И ОТПОРНОСТ НА ОТКАЗЕ

ВЕЖБЕ 4

## Vercel

**Vercel** је платформа за хостовање апликација у *cloud*-у, оптимизована за рад са *JavaScript* радним оквирима као што су *React*, *Next.js*, *Vue* и *Svelte*. Платформа омогућава развој, тестирање и постављање апликација на једноставан и брз начин. Једна од главних предности **Vercel**-а је **автоматизовани CI/CD** процес, при чему се свака измена у коду аутоматски примењује.

Технолошки, **Vercel** користи глобалну **Content Delivery Network (CDN)** која омогућава експресно сервирање садржаја без обзира на географску локацију корисника. Подржава и **serverless** функције, што значи да се може имплементирати **backend** логика без потребе за класичним сервером. Ово знатно поједностављује развој и одржавање апликација.



Платформа такође подржава преглед **deploy**-ја за сваку грану или **pull request**, чиме се тимовима омогућава лакше тестирање и валидација. Интеграција са **Git** системима је беспрекорна, а конфигурација окружења (**environment variables**) је интуитивна. Захваљујући скалабилној инфраструктури, **Vercel** је постао добар избор за многе пројекте који захтевају брзину, флексибилност и отпорност у производњи.



# РЕПЛИКАЦИЈА И ОТПОРНОСТ НА ОТКАЗЕ

ВЕЖБЕ 4

## Задатак

Креирати налог на **Vercel** платформи помоћу постојећи **GitHub** налога. Истражити могућности које **Vercel** нуди, као и видети како повезати репозиторијум са **React** апликацијом на **Vercel**.



# НАДЗОР И УПРАВЉАЊЕ АПЛИКАЦИЈОМ

ВЕЖБЕ 5



## Увод

У дистрибуираним системима, **надзор и управљање апликацијом** представљају кључне активности које омогућавају одржавање **стабилности, поузданости и перформанси** система. Како се компоненте апликације извршавају у различитим окружењима и локацијама, важно је имати централизован и доступан увид у рад система у реалном времену. **Vercel**, као платформа за **deployment** апликација, обезбеђује читав низ алата за надзор, управљање и одржавање апликација у производном окружењу.

Једна од централних особина **Vercel**-а је што се све компоненте апликације – од корисничког интерфејса до **API** функција – извршавају у **cloud**-у. То омогућава континуирану анализу и праћење свих активности, без потребе за додатном инфраструктуром. Сваки **deployment** на **Vercel**-у је верзионисан и може бити независно тестирана инстанца (нпр. **preview deploy**), што омогућава безбедно управљање изменама и праћење њиховог утицаја.

**Vercel** пружа увид у метрике као што су време одговора (**response time**), статус **HTTP** захтева, **латенција**, и успешност извршавања функција. Уграђени аналитички систем омогућава брзо откривање проблема попут спорог учитавања, отказаних функција или грешака у комуникацији са спољним **API**-јима.

## Потребан софтвер

- **Visual Studio Code**  
<https://code.visualstudio.com/download>
- **Node.js**  
<https://nodejs.org/en/download>

## Питања

- Шта је **Vercel** и коју улогу има у дистрибуираном систему?
- На које начине је могуће **deploy**-овати апликацију на **Vercel**?
- Које предности има **deployment** преко GitHub интеграције?
- Шта су **Vercel templates** и у којим ситуацијама се користе?
- На који начин **Vercel** омогућава надзор над **serverless** функцијама?
- Објасни улогу **CDN**-а у контексту **Vercel** платформе.
- Шта су променљиве окружења (**env variables**) и како се конфигуришу на **Vercel**-у?
- Објасни концепт **rollback** у контексту **deployment**-а на **Vercel**-у.
- Који типови **извештаја** (логова) се могу прегледати у **Vercel Dashboard**-у?
- Како се прате **HTTP захтеви** и **одговори** у оквиру **Vercel serverless** функција?
- Које **метрике** су доступне преко **Vercel**-а ради надзора перформанси?



# НАДЗОР И УПРАВЉАЊЕ АПЛИКАЦИЈОМ

ВЕЖБЕ 5

## Deployment апликације на Vercel

**Deployment** апликације на **Vercel** представља један од најједноставнијих и најинтуитивнијих процеса у модерном веб развоју. Платформа је осмишљена тако да омогући брз, аутоматизован и непрекидан ток од развоја до продукције, без потребе за конфигурисањем сервера, виртуелних машина или **CI/CD** алата.

Основни начин **deployment**-а је повезивање **Vercel** налога са **Git** репозиторијумом, било да је он на **GitHub**-у, **GitLab**-у или **Bitbucket**-у. Након повезивања, сваки нови **commit** у главној или било којој грани аутоматски покреће процес изградње (**build**) и **deploy**-а. За главну грану (нпр. **main** или **master**), резултат је **продукциона** верзија апликације, док све остале гране добијају засебне **preview deploy**-eve. Ово омогућава преглед и тестирање сваке измене пре њене интеграције у продукцију.

The screenshot shows the Vercel web interface. On the left, under 'Import Git Repository', there's a dropdown for 'owlCoder' and a search bar. Below it is a list of repositories: 'ad-spisak-api' (12/20/24), 'ad-spisak-reactjs' (Jun 18), 'ad-spisak-xlsx-api' (11/25/24), 'currency-exchange-api' (11/19/23), and 'oib-api' (12/5/23). Each item has an 'Import' button. At the bottom is a link 'Import Third-Party Git Repository →'. On the right, under 'Clone Template', there are four cards: 'NEXT.js Boilerplate' (with a preview of a weather app), 'AI Chatbot' (with a preview of a weather app), 'Commerce' (with a preview of a shopping cart), and 'Vite + React Starter' (with a preview of a weather app). At the top right of this section is a 'Framework' dropdown set to 'React'.

Поред **Git** интеграције, **Vercel** нуди и **deployment** преко предефинисаних **шаблона**. Корисници могу одабрати шаблоне (**templates**) за различите радне оквире као што су **React**, **Next.js**, **Vue**, **Svelte** итд. и тиме креирали нови пројекат директно из **Vercel** интерфејса.

Цео процес **deployment**-а на **Vercel**-у подразумева:

1. Аутоматско откривање радног оквира и креирање оптималне **build** команде;
2. Конфигурацију **public** директоријума;
3. Додавање променљивих окружења (**env**);



# НАДЗОР И УПРАВЉАЊЕ АПЛИКАЦИЈОМ

ВЕЖБЕ 5



## Deployment апликације на Vercel

Управљање евиденцијом догађаја (логовање) је кључни аспект сваке производне апликације. У контексту **Vercel**-а, ово се односи пре свега на праћење извршавања **serverless** функција, **HTTP захтева и грешака** које се могу појавити током рада апликације. Систем за логовање на **Vercel**-у је дизајниран да буде прегледан, ефикасан и лак за интеграцију са спољним алатима.

Свака функција која се извршава у оквиру **Vercel** платформе креира запис у извештају:

- време извршавања,
- HTTP статус,
- излазне поруке (`console.log`, `console.error`),
- време покретања и завршетка функције,
- *event input payload* (ако се не маскира ради безбедности).

The screenshot shows the Vercel Dashboard interface. On the left, there is a list of logs with columns for Time, Status, Host, Request, and Messages. A specific log entry for a POST /api/graphql request at 23:00:52.53 is highlighted. On the right, a detailed view of this log entry is shown in a modal window. The details include the request ID, path, host, user agent, firewall status, function invocation details, route, location, runtime, execution duration, and memory usage.

Time	Status	Host	Request	Messages
JUN 22 23:00:52.53	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:52.18	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:51.94	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:51.67	GET 200	quiz-web-sage.vercel.app	/_.next/data/ieKvnVeWkQFz743dKRueW/quizzes/play/1c...	
JUN 22 23:00:49.48	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:49.22	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:48.90	GET 200	quiz-web-sage.vercel.app	/_.next/data/ieKvnVeWkQFz743dKRueW/quizzes/1c2bulZ...	
JUN 22 23:00:45.10	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:44.80	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:44.42	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:44.05	GET 200	quiz-web-sage.vercel.app	/_.next/data/ieKvnVeWkQFz743dKRueW/quizzes.json	
JUN 22 23:00:42.63	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:42.46	GET 200	quiz-web-sage.vercel.app	/_.next/data/ieKvnVeWkQFz743dKRueW/index.json	
JUN 22 23:00:41.75	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:41.62	GET 200	quiz-web-sage.vercel.app	/_.next/data/ieKvnVeWkQFz743dKRueW/login.json	
JUN 22 23:00:39.70	POST 200	quiz-web-sage.vercel.app	/api/graphql	
JUN 22 23:00:31.18	GET 200	quiz-web-sage.vercel.app	/_.next/data/ieKvnVeWkQFz743dKRueW/login.json	
JUN 22 23:00:02.38	GET 200	quiz-web-sage.vercel.app	/_.next/data/ieKvnVeWkQFz743dKRueW/login.json	
JUN 22 22:59:59.59	POST 200	quiz-web-sage.vercel.app	/api/graphql	

Ови логови су доступни у реалном времену преко **Vercel Dashboard**-а, где се за сваки **deployment** може прегледати појединачна функција и њено понашање. Ово омогућава брзо откривање проблема, одговор на грешке и анализу перформанси.