Local mitone email freste?

Har component chejori neveshte shode tag?  
  
[Google Gemini](https://gemini.google.com/app/4f1cf4c6e1e929db)

1 of purpose of npm init -y is to create a package.json file in your project's root directory.

~~npm install react react-dom parcel --save-dev ????????????~~

npm install react react-dom

(Parcel is a zero-configuration web application bundler - simplifies React development without set up Webpack).

To start development server, run: npm run dev <http://localhost:5173/> har bar taghiri midi bezan update mikone

Run the development server: npm start usually at ~~http://localhost:1234~~, will be live there. (which was parcel public/index.html) is **incorrect** for this Vite setup.

in 2 masir base 2 ta bundler mokhtalefe : parcel and vite

For a Vite project, index.html file includes a script tag that points to your main JavaScript/JSX entry file, like this:

HTML

<body>

<div id="root"></div>

<script type="module" src="/src/ index.jsx"></script> </body>

**deploy** your website to a web hosting service. create a production build, you should **terminal** and navigate to project directory

npm run build

Vite will compile all React components, JavaScript, CSS, and other assets into a highly optimized set of static files (HTML, CSS, JavaScript, images). placed in a folder dist ( distribution") inside project root. upload it to web.

**Step 2: Choose a Hosting Platform (Using GitHub Pages)……..**

[Create a new project from git | Create a new project from git | Create a new project from git | Netlify](https://app.netlify.com/start/repos/bagherianahita%2Fmy-portfolio-website)

[Deploy details | Deploys | mellifluous-crisp-1bdf18 | Netlify](https://app.netlify.com/projects/mellifluous-crisp-1bdf18/deploys/685fcbffe802b86f3a7ac007) ~~???chera nemiare branch main~~   
chon to terminal bayad avalin bar ba dastore   
 git push --set-upstream origin main  
code ro push mikardam

[bagherianahita/my-portfolio-website](https://github.com/bagherianahita/my-portfolio-website)

**گام ۱: پروژه خود را به یک مخزن Git تبدیل کنید.**در ترمینال VS Code ، مطمئن در ریشه پروژه هستید.

Bash

git init

* **گام ۲: فایل‌های غیرضروری را نادیده بگیرید (ایجاد فایل .gitignore). نوع (Type):** فایل .gitignore یک **فایل متنی ساده (Plain Text File)** است. نوع آن به عنوان یک "فایل پیکربندی" برای ابزار Git تعریف می‌شود. این فایل پسوند ندارد، و به دلیل اینکه با یک نقطه (.) شروع می‌شود، در سیستم‌عامل‌های مبتنی بر Unix (مانند macOS و Linux) به صورت پیش‌فرض مخفی است.
* یک فایل جدید در ریشه پروژه خود به نام .gitignore ایجاد کنید.
* محتوای زیر را در آن قرار دهید و فایل را ذخیره کنید:
* # dependencies
* /node\_modules
* # build output
* /dist
* # other
* .env
* .DS\_Store
* **توضیح:** این فایل به Git می‌گوید که پوشه‌های node\_modules (که حجم زیادی دارند) و dist (که با هر build مجدداً ایجاد می‌شود) را به مخزن GitHub شما اضافه نکند. سرویس میزبانی شما خودش این پوشه‌ها را در زمان دیپلوی ایجاد می‌کند.

**گام ۳: فایل‌های پروژه را به Git اضافه و کامیت کنید.**

* تمام فایل‌های پروژه خود را به Git اضافه کنید:

Bash

git add .

git config --global user.email "bagheri.anahita65@gmail.com"

git config --global user.name "anahita"

* تغییرات را کامیت کنید (ثبت کنید):

Bash

git commit -m "Initial commit: Set up project files"

**گام ۴: یک مخزن در GitHub ایجاد کنید و کد را به آن بفرستید.** وارد خود در GitHub شوید و یک مخزن جدید (New Repository) ایجاد کنید.نامی برای آن انتخاب کنید (مثلاً my-portfolio-website)

A screenshot of a computer

AI-generated content may be incorrect.

. GitHub دستوراتی را نشان "push an existing repository from the command line" کپی و در ترمینال خود اجرا

Bash

git remote add origin https://github.com/YourUsername/your-repo-name.git

git branch -M main

git push -u origin main

**گام ۵: سرویس میزبانی وب را به GitHub متصل کنید.**

* سورس کد شما در GitHub است، به یک سرویس میزبانی وب مانند **Netlify** یا **Vercel** بروید.
* وارد حساب کاربری خود شوید و روی "Add new site" یا "New project from Git" کلیک کنید.
* **در تنظیمات دیپلوی:** Deploy کلیک کنید. خودکار npm install و npm run build را اجرا , dist را در وب منتشر می‌کند.

هر زمان تغییری در VS Code ایجاد و آن را در GitHub push کنید، وب‌سایت شما به صورت خودکار به‌روز می‌شود.

1. **شما تغییرات را در VS Code ایجاد می‌کنید:** مثلاً یک تصویر جدید اضافه می‌کنید یا متن را تغییر می‌دهید.
2. **کد را به گیت‌هاب push می‌کنید:** terminal vs
3. git add .

git commit -m "Update footer and hero with new link"

git push

note dote after add is important

1. **Netlify تغییرات را تشخیص می‌دهد:** Netlify که به مخزن شما متصل است، بلافاصله متوجه push جدید شما می‌شود.
2. **Netlify به صورت خودکار build را اجرا می‌کند:** Netlify

**نتیجه:** شما دیگر نیازی ندارید که هر بار پوشه dist را به صورت دستی آپلود کنید. این فرآیند کاملاً خودکار است و گردش کار شما را به عنوان یک توسعه‌دهنده بسیار آسان‌تر می‌کند.

1. آدرس را از www.linkedin.com/... به **https://www.linkedin.com/...** تغییر دهید.

**مثال:**

**آدرس اشتباه (که به صفحه داخلی هدایت می‌کند):** <a href="www.linkedin.com/in/your-profile" ...>

**آدرس صحیح (که به سایت خارجی هدایت می‌کند):** <a href="https://www.linkedin.com/in/your-profile" ...>

**مشکل اینجاست که یک فرم HTML/React به تنهایی نمی‌تواند ایمیل ارسال کند.**

کدی که شما ارسال کرده‌اید، فقط **ظاهر (Frontend)** فرم را ایجاد می‌کند. برای اینکه ایمیل ارسال شود، اطلاعات وارد شده در فرم باید به یک **Backend (سمت سرور)** یا یک **سرویس خارجی** ارسال شوند تا پردازش شده و ایمیل فرستاده شود.

در حال حاضر، کد شما فاقد موارد زیر است:

1. **مقصد ارسال اطلاعات:** تگ <form> شما فاقد ویژگی action است که مشخص کند اطلاعات فرم باید به کجا ارسال شوند.
2. **منطق ارسال:** هیچ کد JavaScriptی برای گرفتن اطلاعات ورودی و ارسال آن به یک سرور یا سرویس وجود ندارد.
3. **Backend:** شما یک سرور برای پردازش درخواست و ارسال ایمیل (با استفاده از سرویس‌هایی مانند Nodemailer، SendGrid و...) ندارید.

**راه‌حل: استفاده از Netlify Forms**

بهترین و ساده‌ترین راه حل برای یک سایت استاتیک که روی Netlify میزبانی می‌شود، استفاده از قابلیت **Netlify Forms** است. این سرویس به شما اجازه می‌دهد بدون نوشتن هیچ کد Backend، فرم خود را فعال کنید.

**کاری که باید انجام دهید:**

فایل کامپوننت Contact خود را در VS Code باز کنید و تگ <form> را به صورت زیر تغییر دهید:

JavaScript

<form name="contact" method="POST" data-netlify="true">

{/\* ... your form inputs ... \*/}

</form>

**توضیح تغییرات:**

* **name="contact"**: یک نام به فرم شما می‌دهد تا Netlify بتواند آن را شناسایی کند.
* **method="POST"**: مشخص می‌کند که اطلاعات فرم باید با متد POST ارسال شوند.
* **data-netlify="true"**: **این مهم‌ترین بخش است.** این ویژگی به Netlify می‌گوید که این فرم را مدیریت کند.

**مراحل نهایی:**

1. کد را تغییر داده و ذخیره کنید.
2. تغییرات را به گیت‌هاب push کنید (با استفاده از git add .، git commit و git push).
3. پس از موفقیت‌آمیز بودن دیپلوی در Netlify، فرم شما فعال خواهد شد.
4. سپس می‌توانید فرم را روی سایت خود تست کنید. وقتی فرمی ارسال می‌شود، اطلاعات آن را می‌توانید در داشبورد Netlify، بخش "Forms" مشاهده کنید.

با این روش، شما نیازی به ساخت Backend یا راه‌اندازی سرویس ایمیل ندارید.

CSS includes media queries to adaptable screens.

difference between

      <script type="module" src="/src/index.jsx"></script>

and

      <script src="/src/index.jsx"></script>

به طور خلاصه، استفاده از type="module" بخش اساسی توسعه مدرن وب و React است، زیرا امکان سازماندهی کد به صورت ماژولار، جلوگیری از تداخل‌ها و بهره‌مندی از ویژگی‌های پیشرفته‌تر جاوااسکریپت را فراهم می‌کند. حتی اگر Bundler شما در نهایت یک فایل جاوااسکریپت واحد تولید کند، مفهوم ماژول‌ها در نحوه ساختاردهی و نوشتن کد منبع شما بسیار مهم است.

A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer program

AI-generated content may be incorrect.

Move profile.jpg and hero-video.mp4 into public/assets

**Why use <section> and not <div>?**

choice between <section> and <div> comes down to **semantic HTML** (meaningful HTML).

* **<section> tag:**
  + Represents a standalone, self-contained section of content within a document.
  + It's typically used for grouping related content that would logically appear in an outline of the document. Think of it like a chapter in a book, or distinct topics on a webpage.
  + Examples: "About Us," "Contact Information," "News Articles," "Product Features," or in your case, a "Hero Section" and "Resume Section." Each of these is a distinct, thematic grouping of content.
  + **Benefit:** Improves accessibility for screen readers and helps search engines understand the structure and content of your page better (SEO). It gives meaning to the content it contains.
* **<div> tag (Division):**
  + Is a generic container for flow content. It's a "division" with no semantic meaning in itself.
  + It's used when you need to group content purely for styling (e.g., applying CSS) or for scripting purposes (e.g., JavaScript manipulation) and no more specific semantic element is appropriate.
  + Examples: Wrapping a group of elements to apply a specific flexbox layout, or creating a wrapper for a modal dialog.
  + **Benefit:** Provides a flexible way to structure your page for styling without adding unnecessary semantic weight.

**In your Hero component, using <section> is appropriate because:**

* The "Hero Section" is a distinct, introductory part of your portfolio, containing related content (image, title, subtitle, video).
* It would logically be a separate entry in the page's outline.

You still use <div> *inside* the <section> (e.g., <div className="container hero-content">) because that div is likely used purely for styling purposes (like centering content or applying a max-width) and doesn't represent a new, distinct section of content. It's just a generic container for layout within the semantic section.

**How to comment for an <a> tag (and other JSX elements)?**

In JSX (which is what React components return), you use **JavaScript block comments** for commenting within the JSX structure.

The syntax is {/\* your comment here \*/}.

You cannot use HTML-style comments (``) directly inside JSX because JSX is JavaScript XML, not pure HTML. The HTML comments would be treated as plain text within the JSX.

Here's how you would comment for an <a> tag, or any other element in JSX:

JavaScript

import React from 'react';

const Header = () => {

return (

<header className="header">

<nav className="navbar container">

<a href="#" className="logo">Anahita Bagheri Kalhor</a>

<ul className="nav-list">

<li className="nav-item">

{/\* This is a comment for the Home link \*/}

<a href="#hero" className="nav-link">Home</a>

</li>

<li className="nav-item">

{/\* This link navigates to the Resume section \*/}

<a href="#resume" className="nav-link">Resume</a>

</li>

<li className="nav-item"><a href="#projects" className="nav-link">Projects</a></li>

<li className="nav-item"><a href="#about" className="nav-link">About Me</a></li>

<li className="nav-item"><a href="#contact" className="nav-link">Contact</a></li>

</ul>

</nav>

</header>

);

};

export default Header;

**Key points for JSX comments:**

* They are enclosed in curly braces {}.
* Inside the curly braces, they are standard JavaScript block comments /\* ... \*/.
* They can be placed on a separate line or inline, just like regular JavaScript comments.
* Comments written this way are typically stripped out during the build process, so they don't add to your final bundle size.

انتخاب بین <section> و <div> به معناشناسی (Semantic HTML) برمی‌گرده.

**<div> (تقسیم / Division):** مثل یک "جعبه" بی‌نام در صفحه وب . این جعبه‌ها فقط برای اینکه بتوانید با CSS به آن‌ها استایل بدهید یا با جاوااسکریپت کاری انجام دهید، هیچ معنی خاصی برای مرورگر یا موتورهای جستجو ندارند.

* + مثال: چند عنصر را کنار هم بگذارید و به آن‌ها یک پس‌زمینه رنگی بدهید؛ از <div> استفاده می‌کنید.
* **<section> (بخش / Section):** مثل یک "فصل" یا "بخش" مشخص در یک کتاب است. وقتی محتوایی دارید که به یک موضوع خاص مربوط می‌شود و منطقاً می‌تواند یک قسمت جداگانه از صفحه باشد (مثل بخش "درباره ما"، "تماس با ما"، یا "پروژه‌ها")، از <section> استفاده می‌کنید.

مثال: در رزومه‌ی ، بخش "خلاصه"، "تحصیلات" و "تجربیات کاری" هر کدام یک <section> معنی‌دار هستند.

<section> بهبود معناشناسی سند (Document Semantics) و دسترسی‌پذیری (Accessibility) است.

* **<section>:** هر <section> می‌تواند (و معمولاً باید) یک عنوان ( <h1> تا <h6>) داشته باشد که موضوع آن بخش را مشخص می‌کند. به موتورهای جستجو (SEO) کمک تا ساختار/اهمیت محتوای صفحه درک .

**<div>:**عنصر **بدون معنای ذاتی (Non-semantic)** هیچ اطلاعاتی درباره نوع محتوای درون خود به مرورگر یا ابزارهای دیگر نمی‌دهد.

تنها هدف گروه بندی عناصر برای **اعمال استایل (CSS)** یا **دستکاری با جاوااسکریپت** باشد و هیچ عنصر معنایی دیگری مناسب نباشد. به عنوان مثال، divهایی که برای ایجاد Grid یا Flexbox layouts یا برای ایجاد کانتینرهای برای تنظیم عرض محتوا استفاده می‌شوند، کاملاً مناسب هستند.

اگر هیچ تگ معنایی مناسبی وجود نداشت، آنگاه از <div> استفاده کنید.

**کامنت گذاشتن**

در JSX (که ترکیبی از جاوااسکریپت و XML/HTML در React است)، شما نمی‌توانید از کامنت‌های HTML (مثل ``) به طور مستقیم استفاده کنید. در JSX باید از **کامنت‌های جاوااسکریپت** استفاده کنید، اما با یک تفاوت مهم: آنها باید داخل براکت‌های {} قرار بگیرند.  { /\* متن کامنت شما \*/ }

کامنت معمولی جاوااسکریپت /\* \*/ را می‌نویسید.

* کامنت‌های JSX در واقع همان **کامنت‌های Block جاوااسکریپت (/\* ... \*/) هستند که درون یک عبارت JSX (با استفاده از {}) قرار گرفته‌اند**. این به این دلیل است که JSX در نهایت به فراخوانی تابع‌های جاوااسکریپت (مانند React.createElement()) تبدیل می‌شود و باید از قوانین جاوااسکریپت برای کامنت‌گذاری پیروی کند.
* **چرا از `` استفاده نمی‌کنیم؟** چون JSX یک پسوند سینتکسی برای جاوااسکریپت است و نه HTML خالص. اگر کامنت HTML را مستقیماً در JSX بنویسید، JSX آن را به عنوان یک گره متنی (text node) در نظر می‌گیرد و ممکن است در خروجی نهایی ظاهر شود یا باعث خطای پارس (parsing error) شود.

find parts of your code for future changes:

### 1\. Use Clear and Consistent Comments (Especially with Standardized Tags)

This is the most direct way to mark your code.

\* \*\*Be Specific:\*\* Instead of `// Changed this`, write `// TODO: Refactor this component to use context API for state management after authentication is implemented.`

\* \*\*Standardized Tags:\*\* Adopt common conventions that many IDEs and tools recognize:

\* `// TODO:` Used for features that are incomplete or need to be added.

\* `// FIXME:` Used for known issues or bugs that need fixing.

\* `// HACK:` Used for temporary solutions that should be revisited and properly implemented later.

\* `// REVIEW:` Used for code that needs to be reviewed or could be improved.

\* `// NOTE:` Used for important notes or explanations.

\*\*Example:\*\*

```javascript

const Contact = () => {

// TODO: Implement actual form submission logic here (e.g., using fetch or axios)

// Currently, it just prevents default behavior.

const handleSubmit = (e) => {

e.preventDefault();

console.log('Form submitted!');

// FIXME: Add form validation before submission

};

return (

<section id="contact" className="contact-section container">

{/\* HACK: Temporary placeholder for image upload field. Remove after client decision. \*/}

<h2 className="section-heading text-center">Get in Touch</h2>

<form className="contact-form" onSubmit={handleSubmit}>

{/\* ... form fields ... \*/}

<button type="submit" className="submit-btn">Send Message</button>

</form>

</section>

);

};

```

\* \*\*Leverage IDE Features:\*\* Most modern code editors (like VS Code) have extensions or built-in features that can list all `TODO`, `FIXME`, etc., comments in your project. In VS Code, you can often find these in the "Problems" panel or by searching for "TODO".

### 2\. Utilize Your Version Control System (Git)

Git is your best friend for tracking changes over time.

\* \*\*Descriptive Commit Messages:\*\* When you make a change (even a small one), write a clear and concise commit message explaining \*what\* you did and \*why\*.

\* Bad: `fix stuff`

\* Good: `feat(Hero): Update profile image path to /assets/profile.jpg`

\* Good: `refactor(App): Split into reusable components (Header, Hero, etc.)`

\* \*\*Feature Branches:\*\* For larger planned changes, create a new Git branch. This allows you to work on a feature in isolation and easily track all related changes.

\* \*\*`git blame`:\*\* If you remember roughly \*when\* you made a change, `git blame <filename>` will show you line-by-line which commit (and by whom) last modified each line.

\* \*\*`git log`:\*\* Use `git log --grep="keyword"` to search commit messages for specific keywords you used.

### 3\. Modularization and Component-Based Architecture

By breaking your application into smaller, focused components (like you've just done with `Header`, `Hero`, `Resume`, etc.), you inherently make it easier to find code.

\* If you know you want to change something about the contact form, you know to look in `Contact.jsx`.

\* If you want to modify the navigation, go to `Header.jsx`.

\* Well-defined responsibilities for each file/component mean less searching.

### 4\. Meaningful Naming Conventions

This is about "self-documenting code."

\* \*\*Clear Variable and Function Names:\*\* Names should clearly indicate their purpose.

\* Bad: `const x = 5;`

\* Good: `const maxItemsPerPage = 5;`

\* Bad: `function doSomething(data) { ... }`

\* Good: `function processUserData(userData) { ... }`

\* \*\*Consistent File/Folder Naming:\*\* Keep your file structure logical and consistent. If all components are in `src/components/`, it's easy to find them.

### 5\. Code Editor / IDE Features

Your development environment has built-in tools for this:

\* \*\*Search (`Ctrl+F` / `Cmd+F` for current file, `Ctrl+Shift+F` / `Cmd+Shift+F` for entire project):\*\* This is the most basic but powerful tool. If you remember a specific string, class name, or function call, you can search for it.

\* \*\*Bookmarks:\*\* Many IDEs allow you to set bookmarks on specific lines of code. This is great for short-term "I'll come back to this in 10 minutes" scenarios.

\* \*\*Go to Definition / Find All References:\*\* Right-clicking on a variable, function, or component name often gives you options to jump to its definition or find all places it's used.

### 6\. External Documentation (for larger projects/teams)

For very significant or complex changes, or if you're working in a team, external documentation can be invaluable:

\* \*\*Jira/Asana/Trello (Task Management):\*\* Create a specific task for the future change, linking to the relevant files or providing context.

\* \*\*Wiki/README.md:\*\* For architectural decisions or major planned refactors, documenting them in a project wiki or a detailed `README.md` can be helpful.

By combining these practices, you'll create a codebase that is not only easier to maintain for your future self but also for any other developers who might work on it. Start with comments and good Git practices – they offer the most immediate benefits\!

\*\*چگونه تغییرات کد را برای یافتن آسان در آینده مشخص کنیم؟\*\*

### ۱. کامنت‌های واضح

به جای `// این رو عوض کردم`، بنویسید:

`// TODO: این کامپوننت را بعد از پیاده‌سازی احراز هویت، برای مدیریت وضعیت از طریق Context API بازنویسی کن.`

\* \*\*تگ‌های استاندارد: قراردادهای رایج IDEها

\* `// FIXME:`: برای مشکلات یا باگ‌های شناخته‌شده‌ای که نیاز به اصلاح دارند.

\* `// HACK:`: برای راه‌حل‌های موقتی که باید بعداً بازبینی و به‌درستی پیاده‌سازی شوند.

\* `// REVIEW:`: برای کدی که نیاز به بازبینی یا بهبود دارد.

\* `// NOTE:`: برای نکات مهم یا توضیحات.

A computer screen shot of a computer code

AI-generated content may be incorrect.

\*\*مثال:\*\*

\* \*\*استفاده از قابلیت‌های IDE:\*\* بیشتر ویرایشگرهای کد مدرن (مانند VS Code) دارای افزونه‌ها یا قابلیت‌های داخلی هستند که می‌توانند تمام کامنت‌های `TODO`، `FIXME` و غیره را در پروژه شما فهرست کنند. پنل "Problems" پیدا کنید یا با جستجو برای "TODO".

### ۲. از سیستم کنترل نسخه (Git) برای ردیابی تغییرات در طول زمان است.

پیام‌های Commit توصیفی:\*\* وقتی تغییری ایجاد می‌کنید (حتی کوچک)، یک پیام commit واضح و مختصر بنویسید که توضیح دهد \*چه کاری\* انجام داده‌اید و feat(Hero): مسیر تصویر پروفایل را به /assets/profile.jpg به‌روزرسانی کن`

\* \*\*شاخه‌های ویژگی (Feature Branches):\*\* برای تغییرات بزرگ‌تر، یک شاخه جدید در گیت ایجاد کنید. این به شما امکان می‌دهد روی یک ویژگی به صورت جداگانه کار کنید و تمام تغییرات مربوطه را به راحتی ردیابی کنید.

\* \*\*`git blame`:\*\* اگر تقریباً به یاد دارید \*چه زمانی\* تغییری ایجاد کرده‌اید، `git blame <نام\_فایل>` به شما نشان می‌دهد که کدام commit (و توسط چه کسی) آخرین بار هر خط را تغییر داده است.

\* \*\*`git log`:\*\* از `git log --grep="کلمه\_کلیدی"` برای جستجو در پیام‌های commit برای کلمات کلیدی خاصی که استفاده کرده‌اید، استفاده کنید.

### ۵. قابلیت‌های ویرایشگر کد / IDE جستجو (`Ctrl+F` / `Cmd+F` برای فایل جاری، `Ctrl+Shift+F` / `Cmd+Shift+F` برای کل پروژه):\*

\* \*\*بوک‌مارک‌ها (Bookmarks):بوک‌مارک‌هایی را روی خطوط خاصی از کد تنظیم کنید. کوتاه‌مدت "بعداً به این برمی‌گردم" عالی است.

\* \*\*رفتن به تعریف (Go to Definition) / یافتن تمام ارجاعات (Find All References):\*\* با راست کلیک روی یک متغیر، تابع یا نام کامپوننت، اغلب گزینه‌هایی برای پرش به تعریف آن یا یافتن تمام مکان‌هایی که از آن استفاده شده است، به شما نمایش داده می‌شود.

### ۶. مستندات خارجی (برای پروژه‌های بزرگتر/تیم‌ها)

\*Jira/Asana/Trello (ابزارهای مدیریت وظیفه):\*\* یک وظیفه خاص برای تغییرات آینده ایجاد کنید و آن را به فایل‌های مربوطه لینک دهید یا اطلاعات زمینه‌ای را ارائه دهید.

\* \*\*Wiki/README.md:\*\* برای تصمیمات معماری یا بازنویسی‌های عمده برنامه‌ریزی‌شده، مستندسازی آنها در یک ویکی پروژه یا یک فایل `README.md` دقیق می‌تواند مفید باشد.

باشه، متوجه شدم. خروجی ترمینال شما اطلاعات جدید و مهمی را نشان می‌دهد.

**۱. مشکل در دستورات پاک‌سازی:**

شما از دستورات ویندوز (rmdir و del) در یک محیط **MINGW64** استفاده کرده‌اید. MINGW64 یک پوسته (shell) شبیه به Linux/Unix (Bash) است و دستورات ویندوز در آن کار نمی‌کنند. به همین دلیل، کامندها با خطاهایی مانند rmdir: failed to remove... و bash: del: command not found ناموفق بوده‌اند.

* rmdir: failed to remove '/q': No such file or directory و rmdir: failed to remove '/s': No such file or directory: این خطاها به دلیل سینتکس اشتباه دستور ویندوز در محیط Bash است.
* rmdir: failed to remove 'node\_modules': Directory not empty: این خطا نشان می‌دهد که پوشه هنوز پاک نشده است.
* bash: del: command not found: این خطا می‌گوید که دستور del در محیط Bash تعریف نشده است.

**۲. مشکل اصلی جدید در npm run build:**

پس از اینکه npm install را اجرا کردید (که با موفقیت 281 packages را نصب کرد)، دستور npm run build را اجرا کردید و با یک خطای کاملاً جدید روبرو شدید:

Error [ERR\_MODULE\_NOT\_FOUND]: Cannot find module '.../vite/dist/node/cli.js' ...

**توضیح مشکل:**

این خطا به این معنی است که **Node.js نمی‌تواند یک ماژول حیاتی از پکیج Vite را در پوشه node\_modules پیدا کند**. این معمولاً زمانی اتفاق می‌افتد که نصب پکیج‌ها ناقص یا خراب شده باشد. از آنجایی که دستورات پاک‌سازی قبلی شما ناموفق بودند، npm install نتوانست کاملاً یک نصب خراب قبلی را جایگزین کند.

**راه‌حل: نصب تمیز (Clean Install)**

برای حل این مشکل، شما باید پوشه node\_modules و فایل package-lock.json را به صورت کامل و صحیح حذف کرده و سپس پکیج‌ها را مجدداً نصب کنید.

**گام به گام در MINGW64:**

1. **پوشه node\_modules و فایل package-lock.json را با دستورات Bash/Unix حذف کنید.**
   * این دستورات جایگزین rmdir و del هستند و در محیط شما کار می‌کنند.
   * در ترمینال خود، دستورات زیر را به ترتیب اجرا کنید:

Bash

rm -rf node\_modules

rm package-lock.json

* + rm -rf پوشه را به صورت بازگشتی و بدون درخواست تایید حذف می‌کند.

1. **پکیج‌ها را مجدداً نصب کنید.**
   * بعد از اجرای دستورات بالا، دستور زیر را اجرا کنید:

Bash

npm install

* + این دستور تمام پکیج‌های شما را به صورت یک نصب تازه و تمیز دانلود و در node\_modules قرار می‌دهد.

1. **دوباره دستور build را اجرا کنید.**
   * پس از اتمام نصب، دوباره npm run build را امتحان کنید.

بعد از اجرای این مراحل، Vite باید بتواند تمام فایل‌های خود را در node\_modules پیدا کند و فرآیند build را بدون خطای ERR\_MODULE\_NOT\_FOUND ادامه دهد.