9/02/24

Episode - 02 Igniting the app.

→ We need to minify, remove unwanted comments & optimize our code to make it production ready.

→ when we use 'create React App', we already get a Small ready to production react app. & it have npm into it already.

→ There are allot of different packages which comes with react which makes our app fast, Not just react alone.

★ NPM: It is everything but not Node package manager ✗
↓
officially it does not have any full form.

→ Basically NPM → manages the packages ★★★

NPM basically helps in managing & adding different Packages. (Packages are pre-written tested code for a Specific task) into our project. It is a collection of millions of Packages.

→ Millions of Packages exist and there is atleast one package for your task.

→ npm init: → To use npm into our project.
It is like a box for us into which we can arrange or organize all our packages.

→ It basically creates a package.json file where we get all our Packages ^ ~~Packages and~~ informations like name, version etc. organized at a Single place.

→ with npm init we can add npm into our project.

★ Package.Json is configuration for our npm
↓
the way in which the parts of Something, or a group of things are arranged.

→ In package.json we can see all the installed packages & their versions in dependencies section.

→ When we write npm init, it asks us some questions like package name, version etc. So it is also like we are creating a new package.

→ Packages & dependencies are same.

★ Bundler :- It is a node packages which basically bundles our project and make it production ready.

→ Some popular bundlers are webpack, Parcel, Rollup, ~~Vite~~ Vite etc

→ A bundler takes all different files like CSS, JS, HTML etc and combines them in a single organized unit.

→ npm install -D → dev dependency → It is generally required during development

→ Normal dependencies can be also used during production.

★ npm install -D parcel: As bundler combines different files & minifies it, and we do it during production ~~that~~ that's why we are installing it as an development dependency.

→ (^) caret: This is present in version of the dependency, listed in dependencies section.

for ex: ^1.4.2 (^) caret only allows path version to be updated, if any new version comes automatically.

major minor patch

1.4.2 to 1.4.3.

* (~) tilde : In dependencies Section if we have version like ~1.4.
Then it will automatically install the minor updated version only. from 1.4.2 to 1.5.0

* If we doesnot have any Sign nor tilde (~) nor (^) caret, if any updates of that dependencies comes it will not get installed automatically.

* It is recommended to have caret (^) and not tilde (~) because with minor updates allot of things might break into our app.
So always have caret (^) because minor update will not have changes much more than previous one.

10/02/24

Track /

→ package-lock.json :- It keeps the record log of exact version which is installed.

→ package.json : It keeps an approx version of package.

Suppose in package.json we have ^1.8.0 & new update comes as 1.8.5 then our package.json will remain same & 1.8.5 will be reflected into package-lock.json.

→ package-lock.json keeps the record of which version is installed into our project. (exact) (of any package)

→ You may have faced a situation where you say that "my project is working fine on my local storage, but when I deployed it, it is not working !!!"

→ It can may happen because on our local & on the production we have different versions of packages / dependencies.

→ To solve this, package-lock.json have a hash (code) known as integrity, where it keeps track that every dam exact version is being used in production, as it is in my local storage.

→ Node-module :- It is the database or we can say that they are the files of actual codes of all the dependencies or packages which we are using in our project.

→ Transitive dependency :- As we installed parcel, in node modules we can see a file/folder named as parcel, it have all the code of parcel. But along with it we can see allot of additional folder they are the dependencies of parcel.

Parcel can have its own dependencies, dependencies of parcel can also have their own dependencies and much more.

So all these dependencies files are available into node module and this property of dependencies are called Transitive dependency.

→ Every package which we install have ~~their~~ their own Package.Json and in their package.Json their own dependencies/packages are listed. Through which npm will know about the other dependencies and it will automatically install all of them.

→ We don't want to put all node modules into our github or into our production ? → No { when we do ~~git init~~ create react app then we automatically get .gitignore file in which node module is already added. }

→ So add it into .gitignore file, So that git will ignore that file and not upload into ~~github~~.

→ Should I put Package & Package-lock.Json into git ? Yes

Git: A version control System which keeps a track of all the changes which we have made into our project

GitHub: A web based hosting Service for git repositories

→ why you should upload package.Json & package-lock.Json on git ?? Because with help of package.Json & package-lock Json we can recreate our node modules. → Just write npm inst into your terminal.

→ All things which you can re-generate dont upload it into git

★ NPX: used to execute any package.

npx parcel index.html
- parcel → package name
- index.html → Source file of our project

→ after this command parcel make a development build and host it into our local host

→ cdn links are not preferred to bring react into our project.
why ?? → ① Because through cdn we are making another network connection with those cdn links. Instead we can have it in our node module. It will be more easier for us.

② Due to version changes in react, we need to update the cdn links. That's why use npm install react, npm install react-dom

npm i is a short form of npm install

→ But still after installing react & react-dom, our code will not be able to understand what is React & what is ReactDom (browser)

```
const heading = React.createElement("h1", {}, "hiee");
const root = ReactDOM.createRoot(docum... );
```

our browser will not understand these keywords

import React from 'react';
import ReactDOM from 'react-dom';

present in node modules.

→ Also write type = "module" into Script tag into HTML file because, import or export functionality is not available into Simple Java Script.

→ When browsers treats our JS file as ~~Script~~ module then we get features like import or export into our JS file.

☆ Go & read the documentation of Parcel ☆

→ When we write npx parcel index.html : It creates a development build and stores it into dist folder.

→ When we do npx parcel build index.html : It Simplifies / compress & Production ready our app and stores 3 main folders into dist Html, css & Js files and some map files.

→ when we write npx parcel build index.html we get an error

This is because in package.json we have entry point i.e, main set as app.js but we are giving entry point as index.html, so there is a conflict.

To remove this erase map: "app.js" from package.json

☆ Note: We can re-generate ~~regenerate~~, dist & .~~cache~~ .parcel-cache, so no need to upload it to git. So add into .gitignore

→ ★ Browserslist: We can add browserslist into our package.json & add the browser's list, Then our app will make sure that it will run 100% on those browsers. For other rest browsers, it may work or may not.

```
"browserslist": [ "last 2 versions",
                  "last 2 chrome versions" ]
```

"last 2 versions" → It will support to all the last 2 versions of all the browsers present over internet.

An array of list of browsers.

"last 2 chrome versions" → It will support last 2 versions of chrome

→ If you want to run your app in all browsers available on internet then your app may get slower, because due to this there will allot of differential bundles for different browsers.

→ So it will basically make our app more heavy & it will become slow.

★★★★ Note: use browserslist.dev → To check how much % of browsers our app will 100% support.

What is NPM? Not Node Package Manager

Imagine you're building a Lego castle. You need lots of different Lego pieces (bricks, windows, doors, etc.), but wouldn't want to buy each one individually, right?

That's where npm comes in!

It's like a giant **Lego store for code** called "Node Package Manager" (npm). Instead of building everything from scratch, you can browse and download pre-written code packages (like Lego sets) for various tasks:

- **Building user interfaces:** Make your website look pretty with UI libraries like React or Vue.js.
- **Connecting to databases:** Talk to your website's data storage easily with database libraries.
- **Sending emails:** Automatically send emails from your website using dedicated packages.
- **Doing math:** Use advanced math functions without writing complicated code yourself.

Here's why npm is awesome:

- **Saves time:** No need to reinvent the wheel, just use code others have already created and tested.
- **Keeps your code organized:** Packages are self-contained, making your code cleaner and easier to maintain.
- **Large community:** Millions of packages exist, and there's likely one for almost any task you need.

Think of it this way:

- You're the builder, focusing on designing the castle.
- npm provides the Lego sets, pre-built modules to speed up your construction.
- Together, you can build amazing websites and applications more efficiently!

I hope this simple explanation helps!

What is NPM init?

Imagine you're starting a new Lego project and need a special box to organize all your pieces. `npm init` is like that box for your code project!

Here's what it does:

1. **Creates a `package.json` file:** This file is like a label for your project, storing important information like its name, version, and dependencies (the pieces you need).

2. **Guides you through setup:** It asks you questions about your project (name, description, etc.) and fills in the `package.json` file with your answers.

3. **Sets up a basic project structure:** It creates some essential folders and files to get you started.

Why use `npm init`?

- **Saves time:** No need to build the box (file) yourself.
- **Keeps things organized:** All your project info is in one place.
- **Makes collaboration easier:** Shares information with others who want to help build your project.

Remember: You can use `npm init` even if you already have a `package.json` file. It will ask if you want to update it or create a new one.

Bonus tip: If you want to set things up quickly without answering questions, you can use `npm init -y`. It will create a basic `package.json` file with default values.

I hope this easy explanation helps!

All about BUNDLERS

In the world of web development, especially with complex front-end applications, **bundlers** play a crucial role in transforming your code into a format that browsers can understand and execute efficiently. Here's an explanation in the simplest terms:

Imagine you're baking a cake.

- You gather all the ingredients (JavaScript code files, CSS stylesheets, images, etc.), which could be scattered across different bowls and cabinets (representing different files and directories in your project).
- A bundler acts like a **chef** who takes all these ingredients and:
  - **Combines them** into a single, organized unit (think of it as mixing all the ingredients in a large bowl).
  - **Optimizes them** for efficiency (like pre-heating the oven, sifting flour, etc.).
  - **Bakes them** into a delicious cake (transforms the code into a browser-readable format). This final "cake" is what gets delivered to your users' browsers, where they can finally load and display your beautiful website or application.

But why do we need bundlers?

- **Modern applications have many pieces:** Large projects often involve numerous JavaScript files, CSS styles, images, and other assets. Manually managing all these individually would be tedious and inefficient.
- **Browsers have limitations:** Older browsers have trouble loading and processing multiple code files. Bundlers prepare the code in a way that ensures even older browsers can understand it.
- **Optimizations and features:** Bundlers can perform various optimizations (like minification, code splitting) to reduce file size and improve loading speed. Some can also provide additional features like code hot reloading (seeing changes instantly without refresh).

Popular bundlers in the JavaScript world:

- **Webpack:** Versatile and widely used, offering extensive customization options.
- **Parcel:** Simpler setup, suitable for smaller projects and faster development.
- **Rollup:** Focused on creating smaller bundles for libraries and modules.

Remember:

- Using a bundler is essential for building efficient and performant web applications, especially with increasing complexity.
- Choose a bundler that suits your project size, feature needs, and personal preferences.
  I hope this baking analogy helps you understand the magic of bundlers in web development!

Normal Dependencies vs Dev Dependencies

In the world of web development, particularly with React projects, understanding the difference between **normal dependencies** and **dev dependencies** is crucial for managing your project effectively. Here's a simplified explanation:

Normal Dependencies (production dependencies):

- **What they are:** These are packages that your **application directly relies on** to function properly when deployed to a production environment (like live website). Think of them as the essential ingredients needed to bake your cake.
- **Examples:** UI libraries like React, Vue.js, or Angular; routing libraries like React Router; data fetching libraries like Axios; utility libraries like Lodash; and any other packages directly used in your application's core functionality.
- **How they are installed:** Use `npm install <package-name>` or `yarn add <package-name>`.
- **Where they go:** Listed in the `dependencies` section of your `package.json` file.

Dev Dependencies (development dependencies):

- **What they are:** These are packages that are **only needed during development**, not by the final application itself. Think of them as tools the chef uses to bake the cake but aren't part of the cake itself.
- **Examples:** Testing frameworks like Jest or Mocha; linting tools like ESLint or Prettier; build tools like Webpack or Parcel; hot reloading tools like React Hot Loader; and any other packages used for developing, testing, or building your application.
- **How they are installed:** Use `npm install --save-dev <package-name>` or `yarn add --dev <package-name>`.
- **Where they go:** Listed in the `devDependencies` section of your `package.json` file.

Why the distinction matters:

- **Reduces bundle size:** Normal dependencies are included in the final production bundle, so keeping them to a minimum is crucial for faster loading times.
- **Keeps production environment clean:** Dev dependencies aren't needed for the application to run, so excluding them prevents unnecessary files and potential security risks in production.
- **Clear separation of concerns:** Separating dependencies based on their purpose makes your `package.json` file more organized and easier to understand.

Key points:

- **Always consider the intended use:** If a package is essential for running your application, install it as a normal dependency. If it's only needed for development tasks, install it as a dev dependency.
- **Use the appropriate installation flags:** Remember the `--save-dev` flag for dev dependencies.
- **Optimize your `package.json`:** Keeping both sections clean and up-to-date ensures a maintainable and efficient project.

Parcel

- When you run npx parcel index.html, Parcel creates a folder named dist to house the optimized and bundled version of your application's code. It's the final output ready for deployment to a web server.

- It also creates .parcel-cache. In computing, a cache acts like a temporary storage space that holds frequently accessed data or results for quicker retrieval.