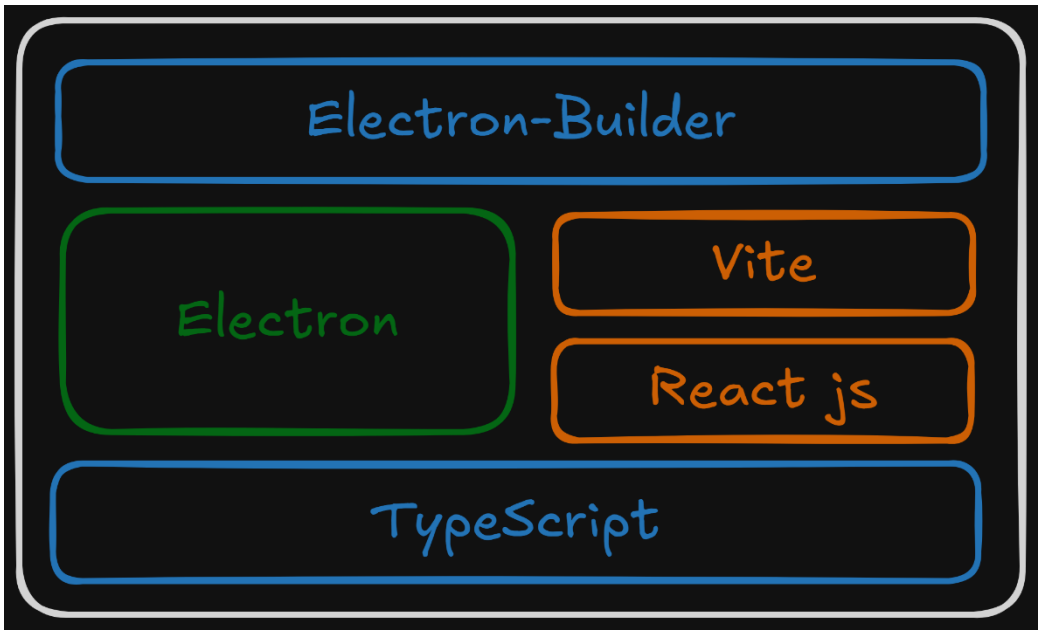
الکترون در ری اکت جی اس (Electron + React js + TypeScipt):



موارد مورد استفاده در این ساختار پروژه :

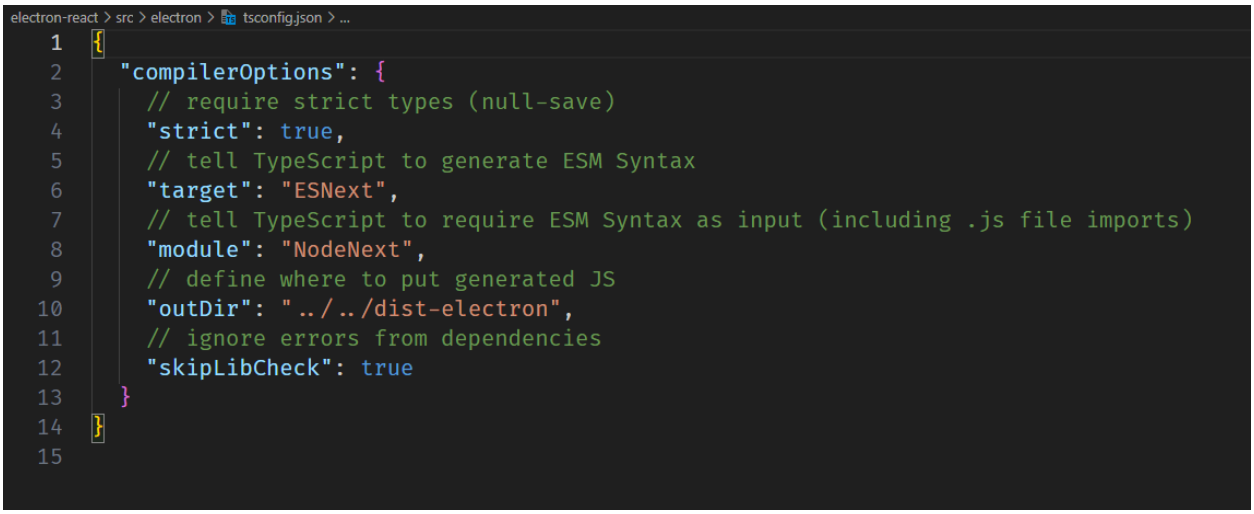
1. TypeScript : در این پروژه برای توسعه از زبان تایپ اسکریپت استفاده شده.
2. Vite: از vite به عنوان یک باندلر به روز برای توسعه و کانفیگ و ارتباط بین تکنولوژی ها و ساختار اصلی پروژه استفاده شده .
3. React js : از این کتابخانه برای توسعه ui پروژه استفاده شده است .
4. Electron : از این فریمورک برای ساخت پروژه اصلی که یک برنامه تحت دسکتاپ به صورت چندسکویی است استفاده می کنیم .
5. Electron-Bulder : در نهایت از این پکیج برای گرفتن خروجی نهایی پروژه برای سیستم عامل های : ویندوز ، مک و لینوکس استفاده می کنیم.

مراحل ایجاد پروژه :

1) با اجرای دستور npm create vite شروع به نصب vite می کنیم و بعد React و تایپ اسکریپت را به عنوان مقادیر انتخابی برای نصب انتخاب می کنیم .
2) بعد از نصب vite به داستور npm i فایل نودماژول را نصب می کنیم .
3) به فایل index.html رفته و مسیر فایل اسکریپت را به "src/ui/main.tsx" تغییر می دهیم .
4) فایل main.tsx و دیگر فایل های پایه مربوط به پروژه react را در پوشه ui که در مسیر src ایجاد می کنیم قرار می دهیم.
5) به فایل vite.config.ts میرویم و مقدار outDir را با dist-react تنظیم می کنیم تا بیلد پروژه ری اکت در این پروژه قرار بگیرد
6) در فایل gitignore نام dist-raect را اضافه می کنیم .
7) یک build از پروژه react می گیریم .
8) با اجرای دستور npm i –save-dev electron فریمورک الکترون را به پروژه خود اضافه می کنیم .
9) به مسیر src باز میگردیم و یک پوشه به نام electron می سازیم وفایل main مربوط به الکترون را ایجاد می کنیم .
10) به فایل package.json میرویم و فایل main را برای پروژه الکترون خود مشخص می کنیم .
11) در قدم بعد اسکریپت های div:react و div:electron را برای اجرای دو پروژه در زمان توسعه مشخص می کنیم.
12) به فایل vite.config.ts میرویم ومقدار base را با "./" مشخص می کنیم تا پروژه در لود فایل های css و js به مشکل نخورد و مجدد یک بیلد میگیریم
13) به فایل tsconfig.json می رویم و به عناون exclude مقدار "src/electron" را مشخص می کنیم تا کامپایلر تایپ اسکریپت از فایل های

مربوط به پروژه الکترون چشم پوشی کند زیرا میخواهیم کانفیگ جداگانه مشخص کنیم.

14) به دایرکتوری مربوط به پروژه الکترون می رویم و فایل tsconfig.json جدید را ایجاد می کنیم و کانفیگ های مربوطه را در آن قرار می دهیم و تایپ فایل main را به .ts تغییر می دهیم.



15) به فایل package.json می رویم و اسکریپت مربوط به کامپایل پروژه الکترون از تایپ اسکریپت به جاوااسکریپت را اضافه می کنیم.

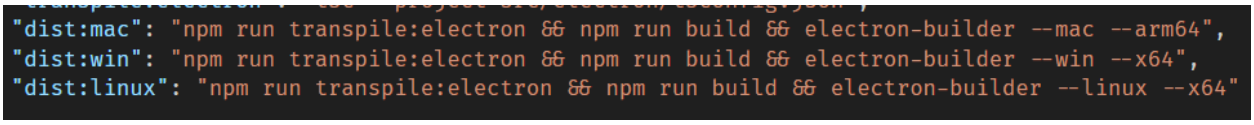
```
"transpile:electron": "tsc --project src/electron/tsconfig.json",
```

16) در فایل gitignore فایل dist-electron را معرفی می کنیم.

17) با اجرای دستور npm i –save-dev electron-builder پکیج مربوط به بیلد نهایی برای سیستم عامل های مختلف را نصب می کنیم.

18) در روت پروژه فایلی با نام electron-builder.json برای قرار دادن کانفیگ های مربوط به بیلد نهایی پروژه می سازیم و کدهای مربوطه را در آن قرار میدهیم.

19) به فایل package.json میرویم و اسکریپت های مربوط به بیلد گرفتن نهایی برای سیستم عامل های مختلف را مشخص می کنیم.



20) برای این که در زمان توسعه با هربار تغییر کد لازم نباشد مجدد از پروژه بیلد بگیریم و این باعث بشه روند توسعه کند و طاقت فرسا بشه باید یک ترفند برای بهینه سازی این روند پیاد سازی کنیم.

21) با اجرای npm i –save-dev cross-env پکیج مربوطه برای خواندن متغییر های محیطی در هر سیستم عامل را نصب می کنیم.

22) اسکریپ dev:electron را با مقدار cross-env NODE_ENV=development electron . تغییر می دهیم.

23) در مسیر پروژه الکترون یک فایل utils.ts می سازیم و در آن یک تابع برای چک کردن اینکه آیا در حالت توسعه هستیم یا نه میسازیم.

electron-react > src > electron > util.ts > ...

```ts
export function isDev(): boolean {
  return process.env.NODE_ENV === 'development';
}

```

24) می توانیم در کانفیگ های vite پورت سرور پروژه ری اکت رو به مقدار دلخواه تغییر دهیم .

electron-react > vite.config.ts > default > server > port

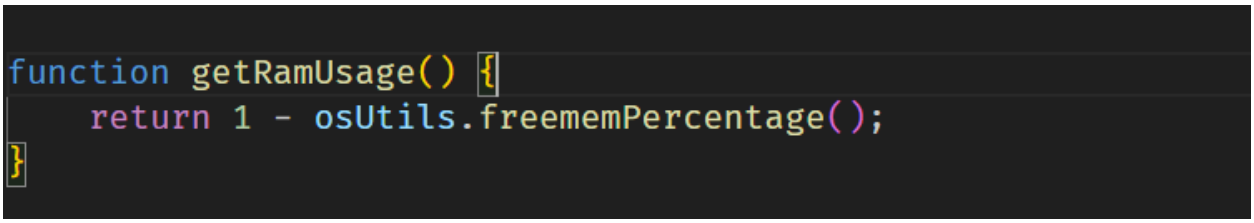
```ts
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  base: './',
  build: {
    outDir: 'dist-react',
  },
  server: {
    port: 5123,
    strictPort: true,
  },
});

```

25) در فایل main.ts مربوط به پروژه الکترون مشخص می کنیم تا در صورت اینکه در حالت توسعه بودیم آدرس لوکال را رندر کند و در غیر این صورت سورس نهایی پروژه را از فایل بیلد شده رندر کند.

26) حالا با هر تغییر در کد ظاهر و ui خروجی را هم در لوکال و هم در اپلیکیشن می بینیم.

27) با اجرای دستور npm i --save-dev npm-run-all میتوانیم پکیج مربوط به اجرای همه دستورات به صورت یکجا را اضافه کنیم. و حالا اسکریپت مربوط به آن را اضافه کنیم و فقط با اجرای یک دستور هم پروژه react و هم پروژه electron را در یک ترمینال اجرا کنیم.

توضیحات مربوط کدهای پروژه :



هدف این تابع محاسبه **میزان RAM استفاده‌شده** از کل حافظه‌ی سیستم است.

تابع freememPercentage() از کتابخانه‌ی os-utils استفاده شده است درصد **RAM آزاد** سیستم را به صورت یک مقدار اعشاری بین 0 و 1بازمی‌گرداند. به عنوان مثال:

- اگر 50% از RAM سیستم آزاد باشد، این تابع مقدار 0.5را برمی‌گرداند.

- اگر 30% از RAM سیستم آزاد باشد، این تابع مقدار 0.3را برمی‌گرداند.

به همین دلیل آن را منهای 1 می کنیم تا میزان رم استفاده شده را بدست بیاوریم.

```
function getStorageData() {
    // requires node 18
    const stats = fs.statfsSync(process.platform === 'win32' ? 'C://' : '/');
    const total = stats.bsize * stats.blocks;
    const free = stats.bsize * stats.bfree;

    return {
        total: Math.floor(total / 1_000_000_000),
        usage: 1 - free / total
    }
}
```

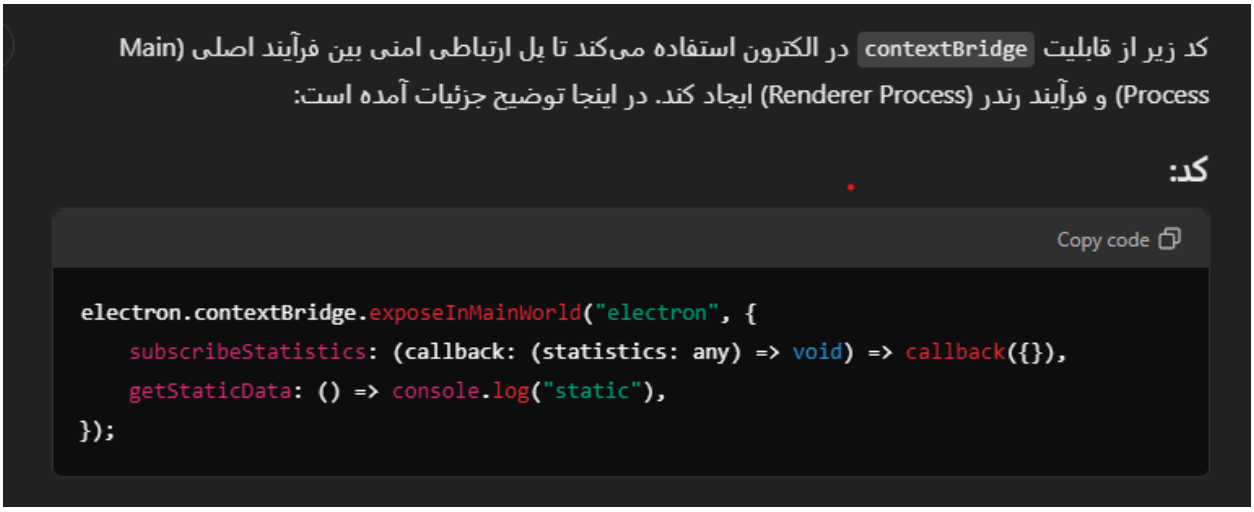
تابع getStorageData برای دریافت اطلاعات مربوط به حافظه‌ی ذخیره‌سازی (مانند ظرفیت کل و میزان استفاده‌شده از دیسک) استفاده می‌شود. بیایید قسمت‌های مختلف این کد را بررسی کنیم:

fs.statfsSync یک تابع همگام از ماژول fs در Node.js (نسخه 18 به بعد) است که اطلاعاتی درباره سیستم فایل (File System) ارائه می‌دهد.

در اینجا، مسیر مورد بررسی بر اساس سیستم‌عامل تعیین می‌شود:

- اگر سیستم‌عامل ویندوز باشد(process.platform === 'win32') ، مسیر درایو ریشه (C://) بررسی می‌شود.
- در سایر سیستم‌عامل‌ها (مانند لینوکس یا مک)، مسیر) /ریشه فایل‌سیستم) بررسی می‌شود.
- **stats.bsize**: اندازه هر بلاک در بایت.
- **stats.blocks**: تعداد کل بلاک‌های حافظه در سیستم فایل.
- با ضرب این دو مقدار(bsize * blocks) ، **ظرفیت کل حافظه‌ی دیسک** به دست می‌آید.
- **stats.bfree**: تعداد بلاک‌های آزاد.

- با ضرب این مقدار در bsize، **ظرفیت فضای آزاد حافظه** محاسبه می‌شود.
- total:
  - ظرفیت کل حافظه دیسک (بر حسب گیگابایت).
  - با تقسیم ظرفیت کل بایت‌ها بر 1,000,000,000 (برای تبدیل بایت به گیگابایت) محاسبه می‌شود.
  - مقدار نهایی با Math.floor گرد می‌شود.
- usage:
  - درصد حافظه‌ی دیسک استفاده‌شده.
  - این مقدار از رابطه زیر محاسبه می‌شود: usage=1−freetotal\text{usage} = 1 - \frac{\text{free}}{\text{total}}usage=1−totalfree
  - total / free درصد فضای آزاد است، و با کم کردن آن از 1، درصد فضای استفاده‌شده به دست می‌آید.



هدف کد:

- **امنیت:** این کد مانع از این می‌شود که فرآیند رندر مستقیماً به APIهای حساس فرآیند اصلی دسترسی داشته باشد.
- **ایزوله‌سازی:** تنها عملکردهای مجاز (مانند `subscribeStatistics` و `getStaticData`) در دسترس فرآیند رندر قرار می‌گیرند.
- **توسعه‌پذیری:** این ساختار قابل گسترش است و می‌توان توابع بیشتری برای ارتباط با فرآیند اصلی اضافه کرد.

توضیحات:

1. `contextBridge.exposeInMainWorld`

این دستور **APIای** را که تعریف شده، به فضای جهانی **Window** در فرآیند رندر اضافه می‌کند. به عبارت دیگر، یک شیء تحت نام `"electron"` در فضای رندر ایجاد می‌شود که شامل دو متد زیر است:

- `subscribeStatistics`
- `getStaticData`

2. `subscribeStatistics`

این یک تابع است که به عنوان آرگومان، یک **کال‌بک** دریافت می‌کند. سپس این کال‌بک بلافاصله با یک شیء خالی (`{}`) فراخوانی می‌شود.
در یک پیاده‌سازی واقعی، این متد ممکن است برای ارسال آمار یا اطلاعات از فرآیند اصلی به فرآیند رندر استفاده شود.