بسم الله العاشق

گزارش کار پروژه

" ایجاد یک وب اپ با پردازش موازی و fastapi "

مربوط به درس پردازش موازی

جناب استاد دکتر رشنو

ریحانه گودرزی

شماره دانشجویی: 40311415012

دانشجوی ارشد نرم افزار

دانشگاه لرستان

بهار و تابستان 1404

|  |  |
| --- | --- |
| فهرستی از مراحل اجرایی پروژه | صفحه |
| 1. کد نویسی پایتون پروژه در دو فاز : | 3 |
| * 1. موازی سازی در سطح نخ ها | 3 |
| * 1. موازی سازی در سطح فرآیند ها | 7 |
| 1. ایجاد یک محیط ماژولار و کاربردی برای اجرا و تست بخش ها و سناریو های متنوع کد های قبل | 11 |
| 1. اجرای مرحله fastapi کردن کل پروژه | 14 |
| 1. ایجاد یک صفحه وب با مشخصاتی که استاد تعیین نموده بودند با ساخت فایل های .HTML و .JS | 17 |
| 1. ایجاد فایل description.pyمربوط به باکس توضیحات هر اجرا در صفحه مرورگر | 19 |
| 1. داکرایز کردن کل پروژه و ایجاد Dockerfile و فایل requirements | 20 |
| 1. ایجاد فایل Nginx و فایل docker-compose.yml | 24 |
| 1. نحوه اجرا (لوکال) | 29 |
| 1. دیپلوی کل پروژه روی یک domain و host | 33 |
| 1. توضیحات تکمیلی | 34 |
| 1. مستندات پروژه | 36 |

|  |
| --- |
| 1. کد نویسی پایتون پروژه در دو فاز : |
| * 1. موازی سازی در سطح نخ ها |

فایل thread\_scenarios.py را میسازیم .

**مقدمه کوتاه :**

در برنامه‌نویسی، نخ (Thread) یک مسیر اجرایی مستقل در داخل یک برنامه است.

با استفاده از نخ‌ها می‌توان چند کار را به صورت **همزمان** یا **شبه‌همزمان** انجام داد.

در پایتون**، ماژول threading** ابزار اصلی برای کار با نخ‌هاست.

گاهی نیاز است نخ‌ها را هماهنگ کنیم **(synchronization)** تا تداخلی در داده‌ها یا ترتیب اجرا ایجاد نشود.

این فایل( thread\_scenarios.py ) شامل ۷ بخش اصلی است که هرکدام چند سناریوی مختلف را نشان می‌دهند. هدف این فایل آموزش و نمایش مفاهیم مختلف کار با نخ‌ها است.

**Section & Scenario :**

🔹 بخش ۱: تعریف نخ (Define Thread)

سناریو ۱: ایجاد چند نخ ساده و اجرای یک تابع در آن‌ها. (آموزش پایه‌ای)

سناریو ۲: نخ‌ها به صورت گروه‌بندی شده اجرا می‌شوند (اول گروه ۱، بعد گروه ۲ و …).

سناریو ۳: نخ‌ها با تأخیر تصادفی اجرا می‌شوند تا ترتیب اجرا تغییر کند.

👉 هدف: آشنایی اولیه با ایجاد و مدیریت نخ‌ها.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: نخ‌ها یکی‌یکی اجرا می‌شوند (ترتیب ثابت).
* سناریو ۲: نخ‌ها در گروه‌های مشخص اجرا می‌شوند.
* سناریو ۳: نخ‌ها با تأخیر تصادفی اجرا می‌شوند (ترتیب غیرقابل پیش‌بینی).

🔹 بخش ۲: نخ فعلی (Current Thread)

سناریو ۱: اجرای همزمان چند تابع ساده در نخ‌های جداگانه.

سناریو ۲: استفاده از current\_thread برای شناسایی نخ فعلی.

سناریو ۳: اجرای نخ‌ها با تأخیرهای مختلف برای شبیه‌سازی شرایط واقعی‌تر.

👉 هدف: یادگیری اینکه هر نخ هویت و نام مستقل دارد.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: هر تابع در نخ خودش اجرا می‌شود.
* سناریو ۲: نام نخ‌ها همزمان با اجرای توابع چاپ می‌شود.
* سناریو ۳: تأخیرهای تصادفی ترتیب شروع و پایان نخ‌ها را تغییر می‌دهد.

🔹 بخش ۳: ساخت زیرکلاس نخ (Subclassing Thread)

سناریو ۱: تعریف کلاس سفارشی نخ و اجرای آن‌ها یکی‌یکی.

سناریو ۲: اجرای نخ‌های سفارشی به صورت موازی و همزمان.

سناریو ۳: نخ‌های زوج با تأخیر شروع می‌شوند تا تفاوت رفتار دیده شود.

👉 هدف: درک قدرت سفارشی‌سازی نخ‌ها با ارث‌بری از کلاس Thread.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: نخ‌های سفارشی پشت‌سرهم اجرا می‌شوند.
* سناریو ۲: نخ‌های سفارشی همزمان اجرا می‌شوند.
* سناریو ۳: نخ‌های زوج با تأخیر شروع می‌شوند.

🔹 بخش ۴: قفل (Lock Synchronization)

سناریو ۱: همه نخ‌ها با قفل هماهنگ می‌شوند تا داده‌ها به هم نریزند.

سناریو 2: اجرای نخ‌ها بدون قفل (امکان تداخل).

سناریو ۳: استفاده از قفل فقط در بخش خاصی از کد (انعطاف بیشتر).

👉 هدف: نشان دادن مشکل «شرایط رقابتی» و اهمیت قفل.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: قفل کل اجرای نخ‌ها را کنترل می‌کند (بدون تداخل).
* سناریو ۲: بدون قفل اجرا می‌شود (امکان تداخل وجود دارد).
* سناریو ۳: فقط بخش شروع نخ‌ها قفل می‌شود (انعطاف بیشتر).

🔹 بخش ۵: قفل بازگشتی (RLock Synchronization)

سناریو ۱: چند نخ روی داده‌ی مشترک کار می‌کنند و با RLock هماهنگ می‌شوند.

سناریو ۲: استفاده تو در تو از RLock (یک نخ می‌تواند چند بار قفل را بگیرد.)

سناریو ۳: تعریف توابع داخلی (add/remove) که هرکدام از RLock استفاده می‌کنند.

👉 هدف: معرفی قفل بازگشتی برای جلوگیری از بن‌بست (Deadlock).

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: نخ‌ها با RLock به داده مشترک دسترسی می‌گیرند.
* سناریو ۲: همان نخ می‌تواند چند بار پشت سر هم قفل را بگیرد.
* سناریو ۳: توابع داخلی add/remove جداگانه RLock را استفاده می‌کنند.

🔹 بخش ۶: سمافور (Semaphore Synchronization)

سناریو ۱: تولیدکننده‌ها (Producers) آیتم تولید می‌کنند و مصرف‌کننده‌ها (Consumers) منتظر می‌مانند.

سناریو ۲: چند تولیدکننده و یک مصرف‌کننده که همه‌ی آیتم‌ها را می‌گیرد.

سناریو ۳: مصرف‌کننده‌ها زودتر از تولیدکننده‌ها شروع می‌شوند و منتظر آیتم می‌مانند.

👉 هدف: مدیریت دسترسی چند نخ به یک منبع محدود.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: تولیدکننده‌ها آیتم تولید می‌کنند و مصرف‌کننده‌ها منتظر می‌مانند.
* سناریو ۲: یک مصرف‌کننده همه آیتم‌های چند تولیدکننده را می‌گیرد.
* سناریو ۳: مصرف‌کننده‌ها قبل از تولیدکننده‌ها منتظر آیتم می‌مانند.

🔹 بخش ۷: مانع (Barrier Synchronization)

سناریو ۱: سه نخ باید در نقطه‌ای مشخص جمع شوند تا همزمان ادامه دهند.

سناریو ۲: یکی از نخ‌ها دیرتر می‌رسد و بقیه منتظر او می‌مانند.

سناریو ۳: مسابقه‌ی چندمرحله‌ای با دو مانع (Barrier) برای هماهنگی در چند گام.

👉 هدف: هماهنگ‌سازی گروهی از نخ‌ها برای شروع همزمان یک مرحله.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: همه نخ‌ها همزمان از یک نقطه (Barrier) عبور می‌کنند.
* سناریو ۲: یک نخ با تأخیر می‌رسد و بقیه منتظر او می‌مانند.
* سناریو ۳: دو مرحله‌ی متوالی با دو Barrier اجرا می‌شود (مسابقه چندمرحله‌ای).

جمع‌بندی :

فایل thread\_scenarios.py به صورت گام‌به‌گام مفاهیم مهم مدیریت نخ‌ها، همزمانی و هماهنگی (Synchronization) را با مثال‌های عملی نشان می‌دهد. هر بخش روی یک مفهوم تمرکز دارد و سناریوها هم برای نمایش تفاوت‌ها طراحی شده‌اند.

|  |
| --- |
| یک مثال از دنیای واقعی برای درک بهتر این قسمت: |
| 🧵نخ‌ها (Threads) تشبیه: بازیکنان یک تیم در یک زمین: همه بازیکنان (نخ‌ها) توی یک تیم هستن و زمین، توپ و مربی (حافظه) رو مشترک دارن. |

بخش ۱ (Define Thread): بازیکنان یکی‌یکی یا با هم وارد زمین می‌شن و بازی می‌کنن.

بخش ۲ (Naming Thread): هر بازیکن شماره پیراهن داره تا مشخص بشه کیه.

بخش ۳ (Daemon Thread): بعضی بازیکنان فقط تا وقتی مربی اصلی توی زمین هست بازی می‌کنن، بعدش بیرون می‌رن.

بخش ۴ (Kill Thread): اگه بازیکنی خطا کنه یا طولانی بازی کنه، داور (برنامه اصلی) می‌تونه اخراجش کنه.

بخش ۵ (Subclass Thread): بازیکنان تخصصی دارن (مهاجم، مدافع، دروازه‌بان).

بخش ۶ (Queue Exchange): بازیکنان پاس (توپ) رو از طریق یک خط پاس مشخص (صف) رد و بدل می‌کنن.

بخش ۷ (Synchronization): نمی‌شه همه بازیکنان همزمان یک توپ رو بزنن؛ باید نوبتی یا هماهنگ باشن.

بخش ۸ (Thread Pool): تیم فقط تعداد محدودی بازیکن می‌تونه توی زمین داشته باشه، بقیه منتظر روی نیمکت می‌مونن.

|  |
| --- |
| .1کد نویسی پایتون پروژه در دو فاز : |
| (bموازی سازی در سطح فرآیند ها |

فایل process\_scenarios.pyرا می سازیم .

این فایل دقیقا مشابه فایل نخ‌ها هست اما اینجا روی پردازه‌ها (Processes) تمرکز کرده.

**مقدمه ساده :**

در پردازه (Process) هر وظیفه روی یک پردازنده جداگانه اجرا میشه و حافظه مستقلی داره.

برعکس نخ‌ها (Threads) که حافظه مشترک دارن، اینجا ارتباط بین پردازه‌ها با صف (Queue)، قفل (Lock)، رویداد (Event) یا سینک (Barrier) انجام میشه.

مزیتش اینه که حتی اگر یکی از پردازه‌ها کرش کنه، بقیه سالم میمونن.

در این فایل، ۸ بخش اصلی هست و هر بخش ۳ سناریو داره که حالت‌های مختلف کار با پردازه‌ها رو نشون میده.

**Section & Scenario :**

🔹 بخش ۱ – ایجاد پردازه (Spawning a Process)

سناریو ۱: ساخت یک پردازه ساده و اجراش.

سناریو ۲: اجرای چند پردازه همزمان (هرکدوم کار جدا انجام میدن).

سناریو ۳: پردازه‌هایی که عملیات خاص (مثل محاسبه توان) انجام میدن.

👉 هدف: نشان دادن روش‌های مختلف ساخت و اجرای پردازه.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: یک پردازه ساده ایجاد و اجرا می‌شود.
* سناریو ۲: چند پردازه همزمان اجرا می‌شوند.
* سناریو ۳: هر پردازه تابع متفاوتی اجرا می‌کند (مثلاً محاسبه مربع).

🔹 بخش ۲ – نام‌گذاری پردازه‌ها (Naming a Process)

سناریو ۱: ساخت پردازه با اسم دلخواه.

سناریو ۲: دادن اسم به چند پردازه و اجرای همزمانشون.

سناریو ۳: دسترسی به اسم پردازه‌ها از سمت پردازه مادر.

👉 هدف: مشخص کردن و مدیریت پردازه‌ها با استفاده از نام سفارشی.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: پردازه با نام سفارشی ساخته می‌شود.
* سناریو 2: چند پردازه با نام‌های مختلف ساخته و اجرا می‌شوند.
* سناریو ۳: نام پردازه‌ها از دید والد قابل دسترسی و بررسی است.

🔹 بخش ۳ – پردازه‌های پس‌زمینه (Background Processes)

سناریو ۱: اجرای پردازه در پس‌زمینه بدون join ( پدر ادامه کارشو میده)

سناریو ۲: اجرای پردازه Daemon که با پایان پدر خودش هم کشته میشه.

سناریو ۳: مقایسه حالت join و no-join برای نمایش تفاوت رفتار.

👉 هدف: بررسی تفاوت بین اجرای پردازه با join، بدون join و Daemon.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: پردازه بدون join اجرا می‌شود و پس از پایان والد، قطع می‌شود.
* سناریو ۲: پردازه Daemon به محض مرگ والد کشته می‌شود.
* سناریو ۳: تفاوت بین پردازه‌های با join (صبر والد) و بدون join (قطع ناگهانی) نشان داده می‌شود.

🔹 بخش ۴ – متوقف کردن پردازه‌ها (Killing a Process)

سناریو ۱: ساخت پردازه‌ای که بی‌نهایت کار میکنه و بعد قطعش میکنیم.

سناریو ۲: پردازه‌ای که در شرایط خاص متوقف میشه.

سناریو ۳: چند پردازه ساخته میشن و بعد همزمان توسط پدر Kill میشن.

👉 هدف: کنترل و پایان دادن دستی به پردازه‌ها.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: پردازه بی‌نهایت بعد از چند ثانیه terminate می‌شود.
* سناریو ۲: پردازه فقط در صورت طولانی شدن به اجبار کشته می‌شود.
* سناریو ۳: چند پردازه همزمان ساخته و سپس همه به اجبار متوقف می‌شوند.

🔹 بخش ۵ – ارث‌بری از کلاس Process (Subclassing Process)

سناریو ۱: پردازه ساده با کلاس شخصی‌سازی‌شده.

سناریو ۲: پردازه‌هایی که ID مخصوص دارن و لاگ میگیرن.

سناریو ۳: پردازه شمارنده که از یک عدد تا عدد دیگه شمارش میکنه.

👉 هدف: پیاده‌سازی منطق اختصاصی با ساخت کلاس‌های پردازه.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: یک کلاس ساده‌ی پردازه اجرا می‌شود.
* سناریو ۲: چند پردازه سفارشی Worker اجرا می‌شوند و هرکدام کار مستقل انجام می‌دهند.
* سناریو ۳: یک پردازه شمارنده ساخته می‌شود که بازه‌ای از اعداد را پردازش می‌کند.

🔹 بخش ۶ – استفاده از صف (Queue)

سناریو ۱: برای تبادل داده تولید یک پیام توسط پردازه فرزند و دریافتش در پردازه پدر.

سناریو ۲: تولید چند آیتم توسط یک پردازه و مصرف توسط پردازه دیگر.

سناریو ۳: ارتباط دوطرفه: فرستادن پیام به پردازه فرزند و گرفتن پاسخ از اون.

👉 هدف: تبادل داده و پیام بین پردازه‌ها.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: تبادل یک پیام ساده بین تولیدکننده و مصرف‌کننده.
* سناریو ۲: ارسال چند پیام و مصرف آن‌ها توسط پردازه دیگر.
* سناریو ۳: تبادل دوطرفه پیام‌ها بین دو پردازه با دو صف مستقل.

🔹 بخش ۷ – همگام‌سازی پردازه‌ها (Synchronization)

سناریو ۱: استفاده از Lock تا پردازه‌ها به صورت منظم به منبع مشترک دسترسی بگیرن.

سناریو ۲: استفاده از Event: پردازه‌ها منتظر میمونن تا سیگنال رویداد از پدر بیاد.

سناریو ۳: استفاده از Barrier: همه پردازه‌ها باید به نقطه مشترک برسن، بعد با هم ادامه میدن.

👉 هدف: هماهنگ کردن پردازه‌ها با ابزارهایی مثل Lock، Event و Barrier.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: Lock باعث می‌شود فقط یک پردازه در لحظه به منبع دسترسی داشته باشد.
* سناریو ۲: Event همه پردازه‌ها را منتظر نگه می‌دارد تا یک سیگنال آزادسازی برسد.
* سناریو ۳: Barrier همه پردازه‌ها را همزمان در یک نقطه متوقف می‌کند و سپس آزاد می‌شوند.

🔹 بخش ۸ – استفاده از Pool (استخر فرایند ها)

سناریو ۱: استفاده از Pool.map برای اجرای همزمان یک تابع روی چند داده.

سناریو ۲: استفاده از Pool.apply\_async برای اجرای غیربلاکینگ (نتیجه بعدا جمع‌آوری میشه).

سناریو ۳: اجرای چند کار با تاخیر و مدیریت‌شده روی پردازه‌ها.

👉 هدف: اجرای کارهای موازی روی داده‌های زیاد با مدیریت خودکار پردازه‌ها.

**رفتار خروجی سناریو های این بخش :**

* سناریو ۱: همه داده‌ها با Pool.map پردازش و نتایج یکجا جمع می‌شوند.
* سناریو ۲: پردازه‌ها با apply\_async به‌صورت غیربلوکی کار می‌کنند و نتایج به مرور جمع‌آوری می‌شوند.
* سناریو ۳: پردازه‌ها به صورت تاخیری اجرا می‌شوند و ترتیب پایانشان به تعداد Workerها بستگی دارد.

**جمع بندی:**

👉 نتیجه اینکه این فایل نشون میده چطور میشه با پردازه‌ها کارهای موازی، همزمان و امن انجام داد و برای هر حالت از ابزارهای مختلف (Queue, Lock, Event, Barrier, Pool) استفاده کرد.

|  |
| --- |
| یک مثال از دنیای واقعی : |
| ⚙️ پردازه‌ها (Processes) تشبیه به تیم‌های مختلف در یک لیگ: هر تیم (پردازه) زمین، توپ، و مربی خودش رو داره و مستقل از بقیه کار می‌کنه. هیچ چیزی بین تیم‌ها مشترک نیست. |

⚙️ پردازه‌ها (Processes) =

بخش ۱ (Spawning Process): تیم‌های جدید برای بازی ساخته می‌شن.

بخش ۲ (Naming Process): هر تیم اسم یا شماره خاصی داره (مثلاً پرسپولیس، استقلال).

بخش ۳ (Daemon Process): بعضی تیم‌ها فقط تا وقتی لیگ فعاله حضور دارن، بعدش حذف می‌شن.

بخش ۴ (Killing Process): لیگ می‌تونه یک تیم رو از ادامه مسابقات محروم کنه.

بخش ۵ (Subclassing Process): بعضی تیم‌ها سبک بازی مخصوص خودشون دارن (تدافعی، هجومی).

بخش ۶ (Queue Exchange): تیم‌ها اطلاعات یا نتایج بازی رو از طریق فدراسیون (صف) به هم منتقل می‌کنن.

بخش ۷ (Synchronization): برای شروع بازی، همه تیم‌ها باید طبق برنامه لیگ با هم هماهنگ باشن.

بخش ۸ (Process Pool): لیگ ظرفیت تعداد مشخصی تیم رو داره و بازی‌ها بین اون‌ها تقسیم می‌شه.

**🔑 خلاصه:**

نخ‌ها = بازیکنان یک تیم (همه چیز مشترک → زمین و توپ).

پردازه‌ها = تیم‌های مختلف در لیگ (همه چیز جدا → زمین و توپ جدا).

|  |
| --- |
| 1. ایجاد یک محیط ماژولار و کاربردی برای اجرا و تست بخش ها و سناریو های متنوع کد های قبل |

فایل test\_runner.py یکی از کلیدی‌ترین بخش‌های پروژه‌ست، چون مسئول اجرا و تست واقعی سناریوها در ترمینال پای چارم است.

ایجاد فایل test\_runner.py

وظیفه اصلی:

این فایل یک رابط اجرایی (runner) برای پروژه‌ست.

کارش اینه که:

* مشخص کنه کدوم بخش (section) و کدوم سناریو (scenario) باید اجرا بشه.
* تفاوت بین Thread و Process رو مدیریت کنه.
* خروجی‌ها و لاگ‌ها رو هندل کنه تا بتونیم در FastAPI استفاده کنیم.

1. **ایمپورت‌ها از فایل‌های دیگه:**

thread\_scenarios → شامل همه‌ی سناریوهای مربوط به Thread

process\_scenarios → شامل همه‌ی سناریوهای مربوط به Process

از پایتون استاندارد:

multiprocessing.Queue و Manager → برای مدیریت صف پیام‌ها در حالت Process

print و لاگ‌ها → برای نمایش خروجی‌ها

1. **تابع اصلی: run\_test()**

def run\_test(section\_number: int, scenario\_number: int, mode: str = "thread", log\_queue=None):

1. **پارامترها:**

section\_number → شماره بخش (۱ تا ۸)

scenario\_number → شماره سناریو (۱ تا ۳)

mode → نوع اجرا (thread یا process)

log\_queue → صف برای ذخیره و ارسال خروجی‌ها در حالت Process

1. **اجرای سناریوها در حالت Thread**

if mode == "thread":

if section\_number == 1:

thread\_section\_1\_define\_thread(scenario\_number)

elif section\_number == 2:

thread\_section\_2\_current\_thread(scenario\_number)

...

بسته به شماره بخش، فانکشن مربوط به اون بخش در thread\_scenarios صدا زده می‌شه.

خروجی‌ها مستقیماً روی stdout چاپ می‌شن.

چون همه‌چیز در یک پردازه انجام می‌شه، نیازی به صف (queue) نیست.

📌 ویژگی: ساده‌ترین حالت اجراست و خروجی رو مستقیم می‌ده.

1. **اجرای سناریوها در حالت Process**

elif mode == "process":

if section\_number == 8:

log\_queue = log\_queue or Manager().Queue()

else:

log\_queue = log\_queue or mpQueue()

اینجا فرق مهمی وجود داره:

برای بخش ۸ (Process Pool) باید از Manager().Queue() استفاده کنیم.

برای بخش‌های ۱ تا ۷ یک multiprocessing.Queue() معمولی کافی است.

📌 چرا؟ چون در Process Pool چند پردازه‌ی فرزند ساخته می‌شه که باید همه بتونن به صف مشترک دسترسی داشته باشن → برای همین از Manager().Queue() استفاده می‌کنیم.

بعد مثل حالت Thread، بر اساس شماره بخش، فانکشن مناسب از process\_scenarios صدا زده می‌شه.

1. **جمع‌آوری و چاپ خروجی‌ها :**

بعد از اجرای هر سناریو در حالت Process :

while not log\_queue.empty():

print(log\_queue.get())

تمام پیام‌هایی که در صف قرار گرفتن، از صف برداشته می‌شن و چاپ می‌شن.

در پروژه‌ی ما این پیام‌ها بعداً توسط WebSocket به مرورگر هم ارسال می‌شن.

1. **بخش تست مستقیم:**

در انتهای فایل:

if \_\_name\_\_ == "\_\_main\_\_":

section = 6

scenario = 3

mode = "process"

run\_test(section, scenario, mode)

این قسمت برای تست محلی نوشته شده.

اگر فایل رو مستقیم اجرا کنی (خارج از FastAPI )، یک سناریوی مشخص رو اجرا می‌کنه.

در این مثال: section=6, scenario=3, mode=process.

📌 این بخش برای دیباگ مفیده، ولی وقتی پروژه داخل FastAPI اجرا می‌شه، این قسمت نادیده گرفته می‌شه.

**جمع‌بندی**

فایل test\_runner.py یک واسط اجرایی است که انتخاب و اجرای سناریوها را مدیریت می‌کند. این فایل مشخص می‌کند که اجرای الگوریتم باید در حالت Thread انجام شود یا در حالت Process. در حالت Thread، خروجی مستقیماً چاپ می‌شود؛ اما در حالت Process، خروجی‌ها در یک صف مشترک (Queue) ذخیره شده و سپس جمع‌آوری و چاپ می‌شوند. این طراحی باعث می‌شود که مدیریت هم‌زمانی در هر دو حالت به صورت جداگانه و شفاف انجام گیرد. همچنین در انتهای فایل بخشی برای تست محلی وجود دارد که امکان اجرای سریع یک سناریو را فراهم می‌کند.

|  |
| --- |
| 1. اجرای مرحله fastapi کردن کل پروژه |

**main.py قلب پروژه**

فایل main.py این فایل اصلی‌ترین نقطه شروع پروژه است.

کارش اینه که:

* برنامه FastAPI رو راه‌اندازی کنه
* روت‌های اصلی (API + WebSocket) رو تعریف کنه
* مدیریت لاگ‌ها و اجرای سناریوها رو هندل کنه
* فایل‌های استاتیک (صفحه وب) رو سرو کنه

1. **ایمپورت‌ها :**

fastapi، HTMLResponse، StaticFiles →

برای ساخت API و سرو فایل‌های استاتیک

CORSMiddleware →

برای رفع محدودیت دسترسی مرورگر به API

run\_test →

تابعی که سناریوها رو اجرا می‌کنه (از فایل test\_runner.py )

descriptions →

توضیحات تئوری الگوریتم‌ها ( از فایل descriptions.py )

logging, Manager, Queue →

برای مدیریت لاگ‌ها و ارسال به WebSocket

WebSocket, WebSocketDisconnect →

برای ارتباط دوطرفه با مرورگر (stream log)

**دقت : نحوه مدیریت لاگ‌ها یکی از نکات خاص پروژه ما 👇**

برای اینکه خروجی Processها (که در یک پردازش جدا اجرا می‌شن) توی مرورگر نشون داده بشه، یک صف اشتراکی (Queue) از multiprocessing.Manager ساخته شده:

manager = Manager()

log\_queue = manager.Queue()

بعد یک Handler اختصاصی به لاگر اضافه شده که هر پیامی که لاگ می‌شه، وارد log\_queue بشه:

class WebSocketLogHandler(logging.Handler):

def emit(self, record):

msg = self.format(record)

self.queue.put(msg)

ویژگی خاص: این طراحی باعث می‌شه خروجی‌ها هم در ترمینال بیاد، هم در WebSocket برای مرورگر ارسال بشه.

1. **تنظیمات FastAPI**

اینجا اپلیکیشن ساخته شده:

app = FastAPI(

title="Parallel Processing API",

description="Run thread/process-based scenarios via FastAPI",

version="1.0"

)

حالا CORSMiddleware اضافه شده تا هر کلاینتی (مثل مرورگر) بتونه به API وصل بشه:

app.add\_middleware( # type: ignore[arg-type]  
 CORSMiddleware,  
 allow\_origins=["\*"],  
 allow\_credentials=True,  
 allow\_methods=["\*"],  
 allow\_headers=["\*"],  
)

مسیر /static برای سرو کردن فایل‌های فرانت‌اند (HTML, JS, CSS) ثبت شده:

app.mount("/static", StaticFiles(directory="static"), name="static")

1. **مسیرها (Endpoints):**

الف) / (صفحه اصلی)

* فایل index.html رو باز می‌کنه و تو مرورگر نمایش می‌ده.
* کاربر با این صفحه کار می‌کنه، نه با API خام.

ب) /run (اجرای الگوریتم)

این یکی از مهم‌ترین قسمت‌هاست.

ورودی‌ها رو از Query Parameters می‌گیره:

section (۱ تا ۸)

scenario (۱ تا ۳)

mode (thread یا process)

بسته به حالت اجرا:

Thread Mode: خروجی مستقیم داخل همون پروسه جمع می‌شه (با StringIO جایگزین stdout) و بعد به کاربر برگردونده می‌شه.

Process Mode: چون خروجی در پروسه جدا هست، مستقیم برنمی‌گرده. فقط یک پیام می‌ده که "Logs will be streamed via WebSocket".

در هر دو حالت، توضیحات تئوری از descriptions.py گرفته می‌شه و همراه با خروجی برگردونده می‌شه.

📌 ویژگی خاص: اینجا دو نوع رفتار متفاوت برای thread و process مدیریت شده.

ج) /ws/logs (وب‌سوکت لاگ‌ها)

مرورگر وقتی حالت Process انتخاب بشه، به این WebSocket وصل می‌شه.

هر وقت چیزی داخل log\_queue بیاد، پیام به مرورگر ارسال می‌شه.

اگر کاربر مرورگر رو ببنده → WebSocketDisconnect هندل می‌شه و ارتباط بسته می‌شه.

📌 ویژگی خاص: این قسمت یک stream real-time log پیاده‌سازی می‌کنه.

**نکات مهم و خاص فایل main.py**

* 1. **Dual Mode Execution:** تفاوت مدیریت Thread و Process خیلی حرفه‌ای هندل شده.
  2. **Queue-based Logging:** استفاده از صف مشترک برای ارسال لاگ‌ها به مرورگر، یک راهکار تمیز برای multi-process logging است.
  3. **WebSocket Integration:** خروجی فرآیندها به صورت زنده به مرورگر می‌ره (نه بعد از پایان کار).
  4. **Static + API Together:** هم فایل‌های فرانت‌اند و هم API در یک اپلیکیشن FastAPI ترکیب شدن (نیازی به سرور جداگانه برای فرانت‌اند نیست).
  5. **Error Handling:** در صورت ورودی نامعتبر (مثلاً شماره سکشن اشتباه)، پیام خطای مناسب برمی‌گردونه.

**جمع‌بندی :**

فایل main.py هسته اصلی پروژه است که اپلیکیشن FastAPI را راه‌اندازی می‌کند. این فایل مدیریت اجرای الگوریتم‌ها در دو حالت Thread و Process را بر عهده دارد. خروجی در حالت Thread مستقیماً جمع‌آوری و بازگردانده می‌شود، اما در حالت Process خروجی‌ها از طریق صف مشترک و WebSocket به مرورگر استریم می‌شوند. همچنین این فایل رابط بین بک‌اند (اجرای واقعی) و فرانت‌اند (صفحه وب) است و با استفاده از مسیر /static فایل‌های سمت کاربر را سرو می‌کند.

|  |
| --- |
| 1. ایجاد یک صفحه وب با مشخصاتی که استاد تعیین نموده بودند با ساخت فایل های .HTML و .JS |

1. **پوشه static/ را ایجاد کردیم**

پوشه **static/** جاییه که فایل‌های **سمت کاربر (Frontend)** قرار داده می‌شوند.

یعنی همون چیزهایی که وقتی کاربر مرورگر رو باز می‌کنه می‌بینه و باهاش تعامل می‌کنه.

1. **فایل index.html**

این فایل همون **صفحه وب اصلی** پروژه است.

* زبانش HTML هست.
* وقتی کاربر وارد http://localhost می‌شه، این فایل لود و نمایش داده می‌شه.
* داخلش یک فرم ساده وجود داره که کاربر می‌تونه باهاش حالت اجرا رو انتخاب کنه:
  + انتخاب بین **Thread** یا **Process**
  + انتخاب **Section** (شماره بخش ۱ تا ۸)
  + انتخاب **Scenario** (شماره سناریو ۱ تا ۳)
* یک دکمه "اجرای الگوریتم" هم داره که وقتی روش کلیک می‌کنیم، درخواست اجرا به بک‌اند (FastAPI) فرستاده می‌شه.

همچنین دو بخش مهم برای نمایش خروجی‌ها وجود دارن:

* **بخش توضیحات** → متن تئوری مربوط به الگوریتم انتخاب‌شده که از descriptions.py میاد.
* **بخش خروجی** → نتیجه واقعی اجرای سناریو که توسط FastAPI و WebSocket ارسال می‌شه.

1. **فایل script.js**

این فایل منطق سمت کاربر (Frontend Logic) رو مدیریت می‌کنه و به HTML جون می‌ده.

* وقتی کاربر روی دکمه اجرا کلیک می‌کنه، این فایل:
  1. مقدار انتخاب‌شده در فرم (mode, section, scenario) رو می‌گیره.
  2. با استفاده از fetch به آدرس /run در FastAPI درخواست می‌فرسته.
  3. **اگر حالت Thread باشه**: خروجی مستقیم از API برمی‌گرده و توی صفحه نمایش داده می‌شه.
  4. **اگر حالت Process باشه**: چون خروجی‌ها باید استریم بشن، یک **WebSocket**  به /ws/logs باز می‌کنه و خط‌ به‌خط لاگ‌ها رو می‌گیره و نشون می‌ده.
* علاوه بر این، کد جاوااسکریپت:
  1. از خطاها محافظت می‌کنه (مثلا اگه API ارور داد، پیام خطا نمایش می‌دهد. )
  2. بعد از هر اجرا خروجی قبلی رو پاک می‌کنه.
  3. اسکرول خروجی رو همیشه به آخرین خط می‌بره (تا کاربر همیشه آخرین لاگ‌ها رو ببینه).

**نقش کلی پوشه static/ در پروژه**

* این پوشه همون **واسط کاربری گرافیکی (GUI)** پروژه‌ست.
* باعث می‌شه هر کاربر بدون نیاز به دستورات ترمینالی، از طریق مرورگر بتونه:
  + الگوریتم‌ها رو انتخاب کنه
  + توضیح هر سناریو رو بخونه
  + خروجی اجرای واقعی رو ببینه

به زبان ساده:

* **main.py + FastAPI** = بک‌اند (اجرای واقعی الگوریتم‌ها)
* **static/ (HTML + JS)** = فرانت‌اند (نمایش و تعامل کاربر)
* **Nginx =**

= واسطه‌ای که این فایل‌ها رو سرو می‌کنه و درخواست‌ها رو بین کاربر و FastAPI رد و بدل می‌کنه.

**جمع‌بندی :**

* پوشه static/ شامل فایل‌های سمت کاربر است.
* فایل index.html صفحه وب اصلی پروژه را تشکیل می‌دهد و رابط کاربری گرافیکی را نمایش می‌دهد.
* فایل script.js منطق سمت کاربر را مدیریت می‌کند: گرفتن ورودی‌ها، ارسال درخواست به FastAPI و نمایش خروجی یا استریم لاگ‌ها.
* این پوشه در کنار FastAPI و Nginx باعث می‌شود پروژه از طریق مرورگر به‌صورت کامل و کاربرپسند اجرا شود.

|  |
| --- |
| 1. ایجاد فایل description.pyمربوط به باکس توضیحات هر اجرا در صفحه مرورگر |

فایل **descriptions.py** در پروژه نقش یک **بانک توضیحات** را دارد.  
در این فایل یک دیکشنری بزرگ به نام descriptions تعریف شده که توضیحات متنی برای هر سناریو را نگه می‌دارد.

* کلید اول این دیکشنری نوع اجرای برنامه است:
  + "thread" → مربوط به سناریوهایی که با **نخ‌ها (Thread)** اجرا می‌شوند.
  + "process" → مربوط به سناریوهایی که با **فرایندها (Process)** اجرا می‌شوند.
* هر کدام از این دو بخش ( thread یا process )خودش یک دیکشنری دارد که **شماره بخش (section)** را به عنوان کلید می‌گیرد.  
  مثلا descriptions["thread"][1] توضیحات بخش اول اجرای thread را برمی‌گرداند.
* در هر بخش هم چند **سناریو (scenario)** وجود دارد (معمولاً ۳ حالت مختلف).  
  پس ساختار نهایی اینطوری است:
* descriptions[mode][section][scenario]

که یک متن توضیحی به زبان ساده برمی‌گرداند.

* این متن‌ها در رابط کاربری وب (صفحه HTML ) نمایش داده می‌شوند تا کاربر وقتی یک سناریو را انتخاب و اجرا می‌کند، بتواند **توضیح تئوری** آن را همزمان کنار خروجی ببیند.

به طور خلاصه:

* **این فایل داده‌ی محض است (Data-Only)** و هیچ تابع یا کدی اجرا نمی‌کند.
* هدفش ما از ایجاد این فایل این است که به صورت **دایرةالمعارف سناریوها** عمل کند: هر بار که کاربر یک سناریو را اجرا می‌کند، سیستم متن مرتبط را از این فایل می‌خواند و نمایش می‌دهد.

|  |
| --- |
| 1. داکرایز کردن کل پروژه و ایجاد Dockerfile و فایل requirements |

Docker چیست؟

یک ابزار که به ما کمک می‌کنه برنامه‌های مان را داخل یک بسته (کانتینر) جمع کنیم تا هرجای دیگه (مثلا در یک سیستم دیگر) آنها را اجرا کنیم ، و بدون مشکل دقیقاً همون‌طور که در سیستم خودمان کار میکند، کار کنند.

برای مثال اگر من یک پروژه که با پایتون (نسخه مورد نظر خودم ) و postgresql نوشتم را با Docker داکرایز (یا کانتینری) میکنم و برای دوستم میفرستم . حالا اون کافیه داکر رو نصب داشته باشه و دستور docker run my-app (اسم پروژه بجای my-app ) را در سیستم خودش اجرا کنه . Docker پروژه من رو در یک محیط ایزوله برای دوستم اجرا میکنه بدون اینکه دوستم نیاز به نصب دستی هیچ چیزی داشته باشه .

درواقع Container (کانتینر) مثل یک مینی سیستم عامل هست که من میتونم پروژه ام رو خیلی سریع و سبک و قابل حمل ، هرکجا که Docker نصب باشه اجرا کنم .

بنابر این داکرایز کردن یعنی آماده کردن یعنی آماده کردن پروژه برای اجرا شدن داخل یک Container .

اجزاء اصلی Docker :

|  |  |
| --- | --- |
| نام | توضیح |
| Image | شِمای کلی پروژه‌ ما ( مثل قالب یا snapshot )  یک قالب آماده که همه چیز موردنیاز اجرای برنامه رو تو خودش داره :  کد، سیستم عامل، ابزارها، تنظیمات ، سیستم‌عامل پایه ( مثلاً Debian یا Alpine ) ، Pythonنصب‌شده ، فایل‌های پروژه، کتابخونه‌های لازم و دستوری که بگه برنامه چطور اجرا بشه  ازش می‌تونی چندین بار کانتینر اجرا کنی |
| Dockerfile | دستورالعمل ساخت Image  Dockerfile یک **فایل متنی ساده** که به Docker می‌گه:  چطور از پروژه‌ات یک Image بسازه؟ چه چیزهایی نصب کنه؟ و چطور اجراش کنه؟  تو با نوشتن یه Dockerfile، به Docker می‌گی :  از چه پایه‌ای شروع کن / چه فایل‌هایی رو کپی کن / چی نصب کن / آخرش چی اجرا کن  و Docker با دستور docker build،  یه Image می‌سازه. |
| Container | نسخه زنده و در حال اجرای همون Image  اجرایImage = ساخت Container  بعد از اینکه Image ساختی، وقتی دستور docker run رو میزنی،  Docker از اون Image یه نسخه زنده می‌سازه به نام Container . |
| Docker Hub | مخزن عمومی برای به‌اشتراک‌گذاری Imageها ( مثل GitHub ولی برای Imageها ) |

حالا میریم سراغ داکرایز کردن پروژه fastapi مون و در کنارش اضافه کردن nginx . برای این کار 4 تا فایل باید بسازیم . مراحلی که باید طی کنیم به شرح زیر است:

1. ساختن requirements.txt
2. نوشتن Dockerfile برای FastAPI
3. نوشتن docker-compose.yml شامل fastapi-app و nginx
4. آماده کردن nginx.conf برای reverse proxy و سرو فایل‌های استاتیک (این مرحله در ادامه توضیح داده میشه)
5. و در آخر، توضیح اجرای لوکال و دیپلوی روی سرور (این مرحله در ادامه توضیح داده میشه)
6. **ساختن فایل requirements.txt :**

این فایل لیستی از تمام کتابخانه ها و پکیج های پروژه ی ما رو نگهداری میکنه . از روی کدهایی که داریم ، نیاز داریم به این پکیج‌ها:

fastapi

uvicorn[standard]

توجه : ماژول‌هایی مثل multiprocessing, threading, asyncio, logging, sys, io, queue همگی جزو کتابخونه‌های استاندارد پایتون هستن و نیازی به نصب ندارن بنابراین نیازی نبود در این فایل اسم اونها رو قرار بدهیم. )

ساخت این فایل مهمه چون بعدا فایل Dockerfileما از این فایل برای کپی خودکارفایل های مورد نیاز پروژه و نصب پکیج ها استفاده خواهد کرد.

نکته اضافی : ما میتونیم نسخه ی دقیق پکیج ها را هم توی این فایل مشخص کنیم این باعث میشه در آینده با تغییر نسخه پکیج ها پروژه به مشکل نخوره (این مرحله اختیاری هست)

fastapi==0.110.1

uvicorn==0.27.1

1. **ساخت فایل Dockerfile:**

گفتیم این فایل Image اپلیکیشن FastAPI رو برامون می‌سازه . این مراحل رو برای ساختش دنبال میکنیم :

توی PyCharm یا File Explorer، یک فایل جدید بساز با نام دقیق: Dockerfile

محتوای داکر فایل رو مینویسیم : مثلا برای یک برنامه ساده hello world :

**مرحله 1: انتخاب تصویر پایه (Python) (**می‌گه Image از کجا شروع شه)

FROM python:3.10-slim

**مرحله 2: تنظیم مسیر کاری داخل کانتینر**

WORKDIR /app

**مرحله 3: کپی کل فایل‌های پروژه به داخل کانتینر**

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

**مرحله 4: اجرای برنامه (**وقتی کانتینر اجرا شد، این دستور اجرا می‌شه**)**

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]

(داکر فایل پروژه من در فایل پروژه م موجود هست )

حالا باید با چند دستور ترمینالی ایمیج رو بسازیم و کانتینرها رو اجرا کنیم.

1. **ساخت docker image پروژه پایتونی مون از روی این dockerfile :**

برای این کار 2 روش وجود داره :

**روش اول: مستقیم با Docker**

اگر فقط Dockerfile رو بخوای اجرا کنی ( بدون nginx و compose )

**روش دوم: با docker-compose (ترجیحی)**

باید فایل docker-compose.yml رو هم بسازیم

⚠️ نکته مهم : بعد از نصب نرم افزار Docker Desktop قبل از انجام مرحله سوم ( با هر روش که انتخاب میکنیم) حتما باید مطمئن بشیم که Docker Desktop در حال اجرا هست . به این طریق:

1. مطمئن شو که Docker Desktop در حال اجراست: از تسک‌بار (پایین گوشه‌ی ویندوز) مطمئن شو آیکون نهنگ 🐳 هست و تول‌تیپش بنویسه: Docker Desktop is running

اگه نیست، از دسکتاپ روی آیکون Docker Desktop دوبار کلیک کن تا اجرا شه.

1. مطمئن شو backend لینوکس داره استفاده می‌شه: Docker Desktop می‌تونه بین دو حالت کار کنه:

Linux containers ✅ ( پیش‌فرض و معمول )

Windows containers ❌

پس روی آیکون نهنگ در تسک‌بار راست‌کلیک کن. اگه نوشته: Switch to Linux containers... یعنی الآن داری از Windows containers استفاده می‌کنی ❌ 👉 پس روش کلیک کن تا سوییچ کنی به Linux containers ✅

اگر این نکته رو دقت نکنی ممکنه موقع اجرای دستورات مرحله 3 چنین خطایی بگیری :

⚠️ خطا :

error during connect: Head "http://%2F%2F.%2Fpipe%2FdockerDesktopLinuxEngine/\_ping": open //./pipe/dockerDesktopLinuxEngine: The system cannot find the file specified.

به زبان ساده یعنی:

«نمی‌تونم به Docker engine متصل شم چون روشن نیست یا در دسترس نیست.»

**روش اول: مستقیم با Docker**

اگر فقط Dockerfile رو بخوای اجرا کنی ( بدون nginx و compose )

(دستورات ترمینالی)

# ساخت ایمیج

docker build -t my-fastapi-app **.**

# اجرای کانتینر

docker run -d -p 8000:8000 --name fastapi-app fastapi-app

بعدش اپت روی <http://localhost:8000> بالا میاد.

# ساخت ایمیج با .... docker build

توضیح:

-t my- fastapi -app: اسم دلخواه برای ایمیج (تو می‌تونی هر اسمی بذاری)

**.**  یعنی داکر بیاد فایل‌های همین پوشه رو بررسی کنه) جایی که Dockerfile هست(

✅اگر همه چیز درست باشه، خروجی‌هایی شبیه این می‌بینی:

Sending build context to Docker daemon ...

Step 1/3 : FROM python:3.12-slim ...

Successfully built a1b2c3d4e5f6

Successfully tagged my-python-app:latest ✅

# اجرای کانتینر با ....docker run

توضیح:

اگر برنامه‌ت مثلاً فقط این باشه:

**app.py**

print("سلام از داخل داکر! 🐳")

تو ترمینال باید همون پیام رو ببینی.

"سلام از داخل داکر ! 🐳 "

# تست :

در مرورگر یا Postman برو به:

http://localhost:8000

و جواب رو ببین:

{"message": "سلام از FastAPI داخل داکر 🚀"}

توجه : تفاوت بین این دو دستور:

دستور ماندگاری کانتینر مناسب برای

🔸 docker run image-name کانتینر باقی می‌مونه تست‌هایی که نیاز به لاگ یا دیباگ داریم

🔹 docker run --rm image-name بلافاصله حذف می‌شه اجرای سریع، موقت، تمیز و بدون ردپا

مدیریت ایمیج‌ها :

عملیات دستور

لیست ایمیج‌ها docker images

حذف ایمیج خاص docker rmi <image\_id یا name>

حذف همه ایمیج‌های بدون استفاده docker image prune یا docker system prune (برای پاک‌سازی کامل)

مدیریت کانتینرها :

عملیات دستور

دیدن کانتینرهای موجود (فعال) docker ps

دیدن همه کانتینرهای موجود (فعال و متوقف شده) docker ps -a

توقف یک کانتینر docker stop <container\_id یا name>

حذف کانتینر docker rm <container\_id یا name>

حذف همه‌ی متوقف‌شده‌ها docker container prune

مشاهده لاگ‌ها docker logs <container\_id یا name>

ورود به داخل کانتینر (مثلاً bash) docker exec -it <container\_id> bash

**روش دوم: با docker-compose (ترجیحی)**

باید اول فایل docker-compose.yml و nginx.conf رو هم بسازیم

|  |
| --- |
| 1. ایجاد فایل Nginx و فایل docker-compose.yml |

1. **ساختن فایل docker-compose.yml :**

این فایل برای راه‌اندازی هم‌زمان دو سرویس در یک پروژه است:

یک سرویس FastAPI و یک Nginx که به عنوان رِوِرس پراکسی عمل می‌کنه.

#برای بالا آوردن همزمان FastAPI و Nginx :

version: "3.9"

#مشخص می‌کنه که از نسخه ۳.۹ از syntax داکر کامپوز استفاده می‌کنیم. این یکی از نسخه‌های پایدار و رایج هست.

services: #این قسمت تعریف همه ی سرویس‌هایی هست که در این پروژه اجرا می‌شن.

fastapi-app: #سرویس اول:

build: .

# می‌گه داکر باید از دایرکتوری فعلی ( جایی که Dockerfile هست) ایمیج بسازه.

#یعنی Dockerfile توی همون فولدریه که docker-compose.yml هست.

container\_name: fastapi-app

#اسم کانتینری که ساخته میشه، میشه fastapi. این اسم توی docker ps نشون داده میشه.

ports:

- "8000:8000"

#فقط برای استفاده داخلی بین کانتینرهاست (نه برای دسترسی بیرونی).

# می‌گه پورت ۸۰۰۰ رو برای سرویس‌های دیگه( مثل ( nginx قابل دسترسی کن.

restart: always

networks:

#این سرویس عضو شبکه‌ای به نام app-network میشه و کانتینرهای داخل یک شبکه می‌تونن با هم ارتباط داشته باشن

- app-network

nginx: #سرویس دوم :

image: nginx:latest

#می‌گه از آخرین نسخه Nginx موجود در Docker Hub استفاده کن.

container\_name: nginx

#اسم کانتینری که اجرا میشه: nginx

ports:

#پورت ۸۰ روی سیستم ما (host) رو به پورت ۸۰ داخل container وصل می‌کنه.

- "80:80"

#پس وقتی مرورگر رو باز میکنیم و به http://localhost میریم درخواست ها به Nginx می‌رسه.

volumes:

#پیکربندی اختصاصی Nginx پروژه ما رو وارد کانتینر می‌کنه. یعنی داری به Nginx می‌گی «این فایل تنظیمات من رو استفاده کن».

- ./nginx.conf:/etc/nginx/nginx.conf:ro

- ./static:/app/static

depends\_on:

- fastapi-app

#می‌گه این سرویس (nginx) بعد از راه‌اندازی fastapi اجرا بشه.

# اما depends\_on فقط ترتیب رو تنظیم می‌کنه، نه اینکه منتظر سالم بودن FastAPI بمونه.

restart: always

networks:

#اینجا هم Nginx در همون شبکه‌ای هست که FastAPI هم هست، پس می‌تونه با اسم fastapi بهش وصل بشه

- app-network

networks: #تعریف شبکه:

app-network:

driver: bridge

#یک شبکه مشترک بین Nginx و FastAPI برای ارتباط داخلی

#یک شبکه مجازی به اسم app-network تعریف شده با bridge به عنوان درایور.

#تمام کانتینرهایی که عضو این شبکه هستن می‌تونن همدیگه رو با اسم سرویس‌ها شناسایی کنن.

#مثلاً داخل Nginx می‌تونی http://fastapi:8000 بنویسی.

## خلاصه ساده:

| **بخش** | **توضیح** |
| --- | --- |
| fastapi | اپلیکیشن Python شما که با FastAPI نوشته شده |
| nginx | پراکسی معکوس که درخواست‌ها رو به FastAPI می‌فرسته |
| volumes | پیکربندی اختصاصی Nginx رو وارد کانتینر می‌کنه |
| networks | یک شبکه مشترک بین Nginx و FastAPI برای ارتباط داخلی |
| ports | دسترسی از مرورگر به Nginx (و از اونجا به FastAPI) از طریق localhost:80 |

1. **ساختن فایل nginx.conf :**

ما می‌خوایم یک اپلیکیشن FastAPI که با uvicorn روی پورت 8000 اجرا میشه رو از طریق Nginx روی پورت 80 یا 443 (HTTPS)به دنیای بیرون نمایش بدیم.

#برای پراکسی کردن درخواست‌ها به FastAPI و سرو فایل‌های استاتیک

events {}

http {

server {

#این بلاک کل تنظیمات یک سرور مجازی (Virtual Host) رو مشخص می‌کنه.

listen 80;

#به Nginx می‌گه که روی پورت ۸۰ گوش بده ( پورت پیش‌فرض HTTP)

#اگر بخوای HTTPS داشته باشی، باید بنویسی: listen 443 ssl; و SSL cert هم تنظیم کنی.

server\_name localhost;

#مشخص می‌کنه که این پیکربندی برای کدوم دامین هست.

#فقط وقتی کاربر به example.com مراجعه کرد، این بلاک فعال میشه.

#اینlocalhost یعنی برای تست محلیه.

location / {

#می‌گه برای هر درخواستی به مسیر / (ریشه)، این کارها رو انجام بده.

#اگه بخوای فقط یه مسیر خاص مثلاً /api رو هدایت کنی باید بنویسی:

location /api {

proxy\_pass http://fastapi-app:8000;

#می‌گه تمام درخواست‌های ورودی به مسیر/رو بفرست به این آدرس:http://app:8000

#اینجا app نام سرویسیه که توی Docker Compose تعریف کردیم (همون سرویسی که FastAPI توشه)

#یعنی انگار داره می‌گه: «بفرست برای کانتینری به اسم app روی پورت ۸۰۰۰»

#این ۳ خط هدرهایی رو تنظیم می‌کنن که برای امنیت، لاگ‌گیری یا تشخیص IP مفید هستن:

proxy\_set\_header Host $host;

#آدرس هاست اصلی رو منتقل می‌کنه به backend (مثلاً FastAPI بفهمه درخواست از کجا اومده)

proxy\_set\_header X-Real-IP $remote\_addr;

#این IP واقعی کاربر رو به backend پاس می‌ده.

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

#زنجیره‌ای از IPهای درخواست‌دهنده‌ها رو منتقل می‌کنه (در درخواست‌هایی که از چندین پراکسی عبور کرده‌اند).

proxy\_set\_header X-Forwarded-Proto $scheme;

}

#اطلاعات مربوط به پوشه statics و صفحه وب ما

location /static/ {

alias /app/static/;

}

}

}

معنی این تنظیمات:

* وقتی تو مرورگر بزنی http://localhost → Nginx همون index.html رو میده.
* وقتی صفحه‌ت با جاوااسکریپت درخواست /run?... یا /ws/logs بفرسته → Nginx درخواست رو به کانتینر FastAPI می‌فرسته.

## ساختار نهایی پروژه

docker-python-demo/

├── main.py

├──thread\_senarios.py

├──process\_senarios.py

├──test\_runner.py

├──descriptions.py

├──statistic

│ └──index.html

│ └──script.js

├── Dockerfile

├── requirements.txt

├── nginx/

│ └── default.conf

└── docker-compose.yml

حالا که فایل docker-compose.yml و nginx.conf رو ساختیم میتونیم بریم سراغ زدن دستورات کامندی روش دوم یعنی ساخت docker image پروژه پایتونی مون از روی dockerfile

**روش دوم: با docker-compose (ترجیحی)**

چون ما فایل docker-compose.yml نوشتیم، کافیه در ترمینال این دستورات رو بنویسیم:

# ساخت ایمیج‌ها و کانتینرها

docker-compose up -d --build

# چک کردن وضعیت کانتینرها

docker ps

# مشاهده لاگ‌ها

docker-compose logs -f

حالا اپلیکیشن FastAPI پشت Nginx ران میشه.

http://localhost ( پورت 80 → nginx) صفحه HTML رو نشون میده.

http://localhost/run?... هم API رو صدا می‌زنه.

ws://localhost/ws/logs هم WebSocket رو هندل می‌کنه.

|  |
| --- |
| 1. نحوه اجرا (local) |

**مراحل اجرای پروژه داکرایز شده:**

1. **آماده‌سازی محیط**

روی سیستم (یا سرور) مطمئن شو **Docker** و **docker-compose** نصب هست.  
برای تست:

docker -v

docker-compose -v

(.venv) PS E:\Arshad\_1403\term2\Parallel\_processing\home\_works\Final\_Progect\_FastAPI\_APP> docker -v

Docker version 28.3.2, build 578ccf6

(.venv) PS E:\Arshad\_1403\term2\Parallel\_processing\home\_works\Final\_Progect\_FastAPI\_APP> docker-compose -v

Docker Compose version v2.38.2-desktop.1

1. **روشن کردن Docker Desktop**

مطمئن شو Docker Desktop رو اجرا کردی و آیکونش سبز یا Running هست.

1. **رفتن به پوشه پروژه**

فرض کنیم پروژه‌ات داخل پوشه‌ای به اسم parallel\_project هست:

cd parallel\_project

مسیر خیلی مهمه با دستور dir در ترمینال مسیر و پوشه هاتو چک کن و برو روی داکر فایلت راست کلیلک کن و مسیرش رو کپی کن و اینجوری در ترمینال مسیر رو عوض کن : cd “path”

1. **ساخت فایل‌های لازم (اگر هنوز نساختی)**

داخل همین پوشه پروژه باید این فایل‌ها باشه:

* Dockerfile
* requirements.txt
* docker-compose.yml
* nginx.conf
* به‌علاوه فایل کدهایی که قبلاً ساختیم ( main.py, thread\_scenarios.py, … و پوشه static/ )

1. **(اختیاری) پاک‌سازی کانتینر/ایمیج‌های قبلی**

وقتی از قبل چیزی ساخته باشی:

docker-compose down

docker system prune -f

1. **ساخت ایمیج و اجرای کانتینرها (در صورتی که بدون docker-compose میخوای کار کنی )**

docker build -t fastapi-app:latest .

این دستور Dockerfile رو می‌خونه، وابستگی‌ها رو نصب می‌کنه و ایمیج اپلیکیشن FastAPI می‌سازه.

1. **اجرای پروژه با docker-compose**

docker-compose up -d --build

این دستور هم ایمیج می‌سازه، هم کانتینرها رو بالا میاره.

برداشتن و دوباره‌ساختن کامل:

docker-compose down

docker-compose up -d --build --force-recreate

پاکسازی کش‌ها/ایمیج‌های قدیمی(اختیاری)

# docker system prune -a

1. **بررسی وضعیت**

docker ps

docker compose ps

باید دو تا سرویس ببینی:

* یکی برای fastapi-app (API و WebSocket)
* یکی برای nginx (برای وب‌سرور و سرو استاتیک فایل‌ها)

**تا اینجا حالا کانتینرهای FastAPI و Nginx روی سیستم بالا هستن و خودشون روی پورت 80 لوکال‌هاست سرویس می‌دن.**

1. **تست در مرورگر**

صفحه اصلی پروژه:  
👉 <http://localhost>

یک درخواست API نمونه:  
👉 <http://localhost/run?section=1&scenario=1&mode=thread>

[**http://localhost/run?section=1&scenario=1&mode=process**](http://localhost/run?section=1&scenario=1&mode=process)

WebSocket

خودش توسط صفحه HTML وصل میشه

اینجا می‌تونی API‌هات رو با Swagger UI تست کنی!

<http://localhost:8000/docs>

uvicorn main:app –reload

این دستور ترمینالی ما هست برای اینکه اگر بخوای بعداً پروژه رو بدون داکر اجرا کنی (مثلاً برای توسعه)

1. **مشاهده لاگ‌ها (اگر خطا ببینیم هم این رو اجرا میکینم)**

docker-compose logs -f

# همه سرویس‌ها

docker-compose logs -f fastapi-app

# فقط سرویس fastapi-app

docker-compose logs -f nginx

# یا فقط nginx

**چه چیزی باید در لاگ‌ها ببینیم؟**

* لاگ fastapi-app باید چیزی شبیه این داشته باشه:
* INFO: Started server process [xxxx]
* INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
* لاگ nginx بعد از درخواست باید نشان دهد که پراکسی انجام شده یا فایل استاتیک سرو شده.
* اگر docker ps نشان داد که کانتینری زود بسته شده (Exit state)، لاگ‌های آن را با docker logs <container-name> بخوان.

1. **مدیریت سرویس‌ها**

توقف:(**وقتی کارت تموم شد**)

docker-compose down

ری‌استارت:

docker-compose restart

پاکسازی (اگر لازم شد و پاکسازی برای آزاد کردن فضا):

docker system prune -a

دقت : این همه ایمیج و کانتینر بلااستفاده رو پاک می‌کنه (مواظب باش چون همه رو حذف می‌کنه).

در صورت نیاز به ورود به کانتینر برای تست دستی:

docker exec -it fastapi-app /bin/sh

بعد از این مراحل، پروژه ما روی **localhost** پشت Nginx در حال اجراست.  
روی سرور هم همین دستورها رو تکرار می‌کنیم (فقط به جای localhost آی‌پی یا دامنه‌ات رو میزنیم)

**جمع‌بندی:  
فقط این ۳ دستور کل کاریه که نیاز داری:**

docker compose up -d --build # ساخت و اجرا

docker compose ps # بررسی وضعیت

docker compose down # توقف

**توضیح خط‌به‌خط آنچه اتفاق می‌افتد**

* docker-compose up -d --build:
  + فایل docker-compose.yml را می‌خواند.
  + برای سرویس fastapi-app، اگر در docker-compose.yml گفته شده build: .، Docker با Dockerfile موجود در پوشه، ایمیج می‌سازد.
  + سپس کانتینرها را ایجاد و اجرا می‌کند. nginx با پورت 80 روی ماشین میزبان نگاشته می‌شود و درخواست‌ها را به کانتینر fastapi-app:8000 هدایت می‌کند.
* پس از بالا آمدن، درخواست http://localhost به Nginx می‌رسد که از مسیر /static صفحهٔ HTML را سرو می‌کند و درخواست‌های API را به FastAPI می‌فرستد.

|  |
| --- |
| 1. دیپلوی کل پروژه روی یک domain و host |

|  |
| --- |
| 1. توضیحات تکمیلی |

**Nginx چیست؟**

یک وب سرور و reverse proxy است که معمولاً برای سرو کردن سایت‌ها و هدایت درخواست‌ها استفاده می‌شود.

وب سرور: می‌تواند فایل‌های HTML، CSS، JS و تصاویر را مستقیم به مرورگر ارسال کند.

Reverse proxy : جلوی یک یا چند اپلیکیشن مثل FastAPI قرار می‌گیرد و درخواست‌ها را به آن‌ها هدایت می‌کند.

**چرا Nginx را با FastAPI استفاده می‌کنیم؟**

1. هدایت ترافیک و مدیریت درخواست‌ها :

* FastAPI خودش با Uvicorn اجرا می‌شود و می‌تواند مستقیم درخواست‌ها را پاسخ دهد. اما وقتی تعداد کاربران زیاد شود یا بخواهیم HTTPS و چیزهای دیگر اضافه کنیم، Uvicorn به تنهایی کافی نیست. Nginx می‌تواند ترافیک را مدیریت کند.

1. امنیت:

* با Nginx می‌توان فیلترهای امنیتی اضافه کرد، HTTPS راه انداخت و جلوی حملات مستقیم به برنامه را گرفت.

1. استفاده از پورت استاندارد وب:

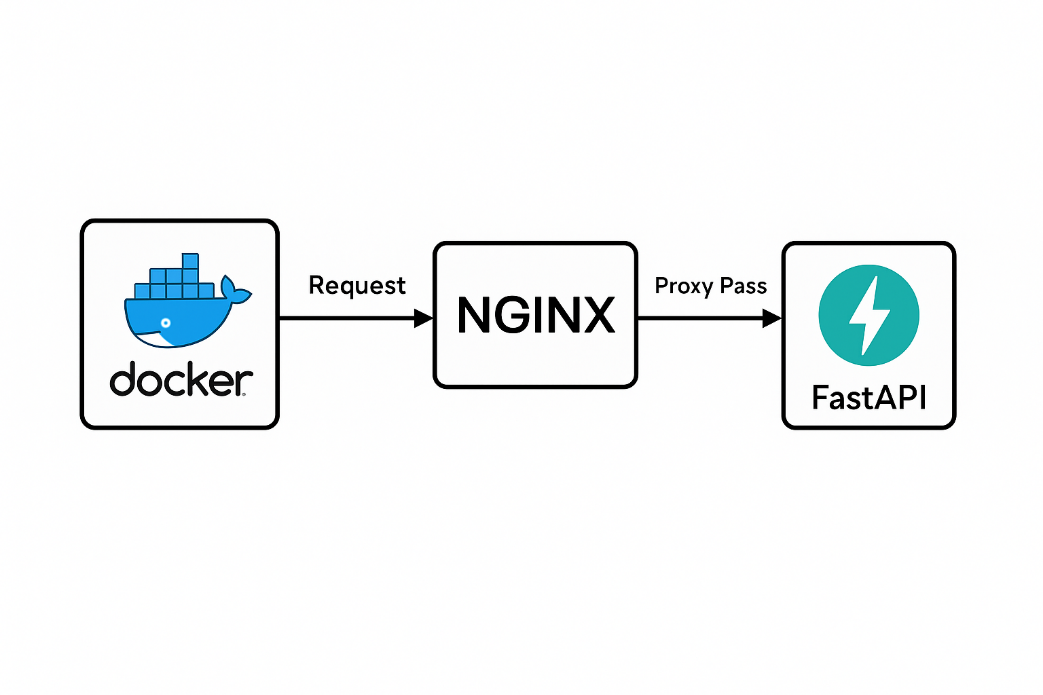
* مرورگرها به طور پیش‌فرض روی پورت 80 (HTTP) و 443 (HTTPS) می‌روند.
* Uvicorn روی پورت 8000 اجرا می‌شود، اما Nginx درخواست‌ها روی پورت 80 دریافت می‌کند و به 8000 هدایت می‌کند.

1. بارگذاری استاتیک سریع‌تر: Nginx

* می‌تواند فایل‌های CSS، JS و تصاویر را خیلی سریع سرو کند، بدون اینکه FastAPI درگیر شود.

**به طور خلاصه:**

FastAPI برنامه شما را اجرا می‌کند، Nginx جلوی آن قرار می‌گیرد و درخواست‌ها را به آن هدایت می‌کند و همزمان کارهای امنیت و بهینه‌سازی را انجام می‌دهد.



این نمودار درواقع بیشتر معماری داخلی کانتینرها را نشان می‌دهد (داخل ماشین یا سرور). اما برای اینکه برنامه‌ ما واقعاً روی اینترنت قابل دسترس باشد باید یک سرور داشته باشیم ( مثلاً VPS، سرور ابری یا حتی یه ماشین فیزیکی)

**جای سرور و مرحله‌ی دیپلوی توی این معماری اینجاست:**

1. توسعه محلی (Local Development):

* کد نویسی روی لپ‌تاپ.
* Dockerfile، docker-compose و nginx.conf رو آماده می‌کنیم.
* تست می‌کنیم که همه‌چیز درست کار کنه.

1. انتقال به سرور (Deploy):

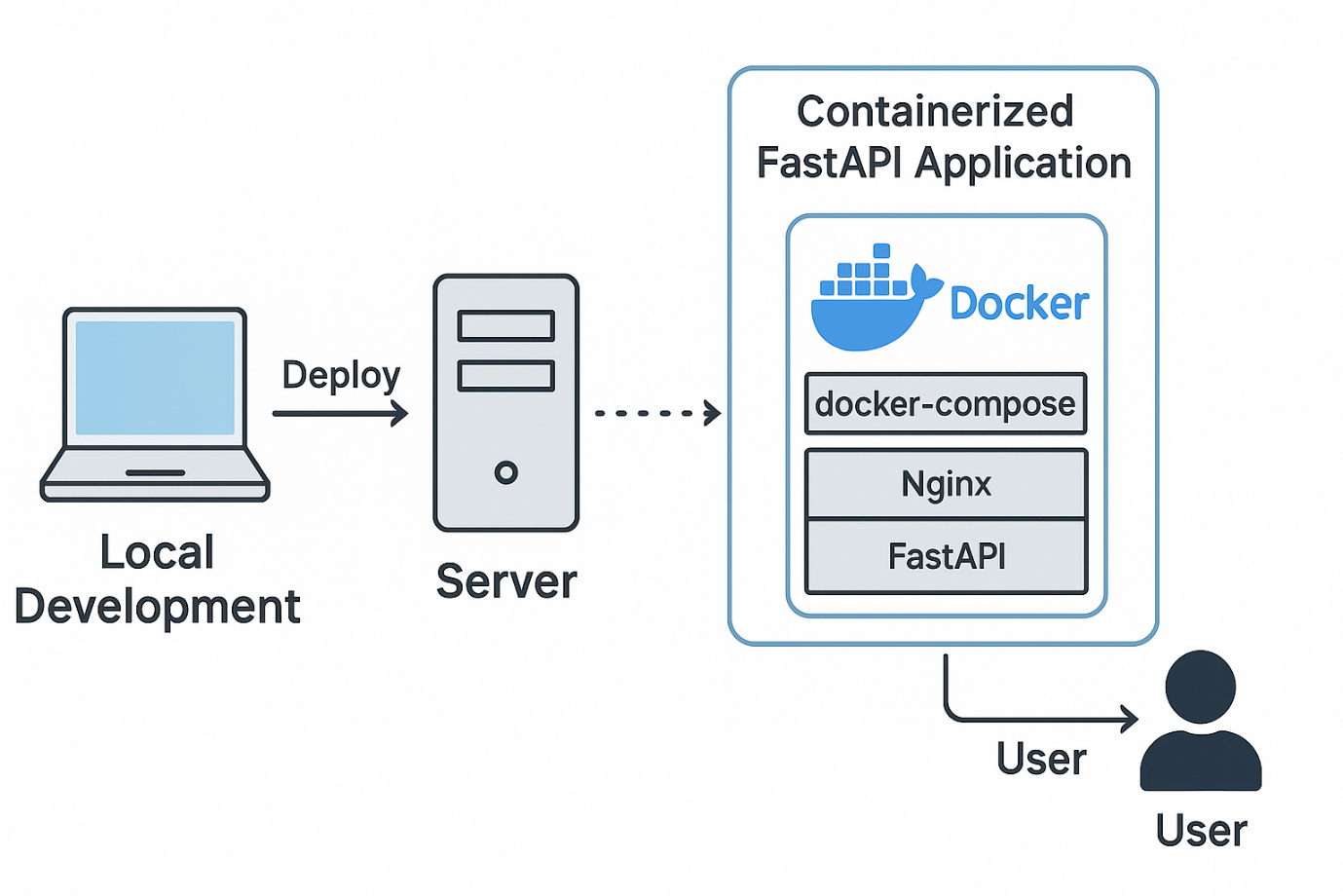
* روی سرور ( مثلاً Ubuntu ) داکر رو نصب می‌کنیم.
* پروژه‌ ( یا فقط Dockerfile و فایل‌های کانفیگ ) را روی سرور کپی می‌کنیم.
* دستور docker-compose up -d یا مشابه اون رو اجرا می‌کنیم.

1. بعد از دیپلوی روی سرور:

* Nginx داخل کانتینر گوش به پورت 80 (HTTP) می‌ده.
* کاربر وقتی آدرس دامنه یا IP سرور ما رو می‌زنه → درخواست میاد روی پورت 80 سرور.
* این درخواست می‌ره داخل کانتینر Nginx → بعد به FastAPI → جواب برمی‌گرده به کاربر.
* یعنی سرور در واقع محیط اجرای همین معماری هست .

به طور خلاصه:

* روی لپ‌تاپ تست می‌کنی
* روی سرور اجراش می‌کنی
* سرور نقطه اتصال بین اینترنت و برنامه‌ ما می‌شه.



|  |
| --- |
| 1. مستندات پروژه |



