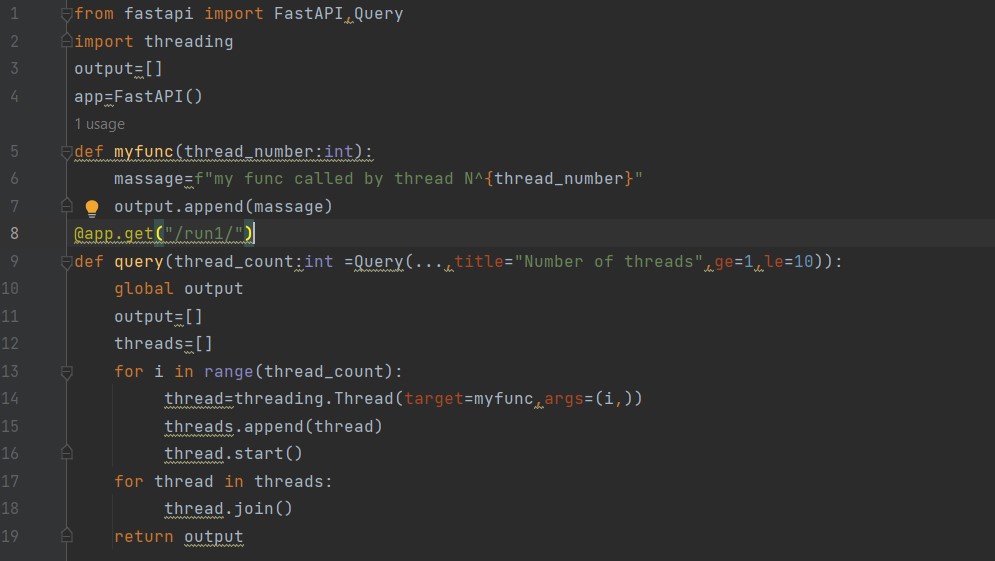
به نام خدا

نوشین نوذری

گزارش پروژه پایانی درس پردازش موازی

تابستان 1403

کارشناسی ارشد مهندسی نرم افزار

Defining a thread:

**1)وارد کردن کتابخانه ها:**

**Fastapi:برای ایجاد و مدیریت APIوب**

**Query:برای تنظیم پارامترها در قالب کوئری**

**Threading:برای مدیریت و ایجاد نخ های همزمان**

**2)تعریف متغیرها و APP**

**Output:لیستی از خروجی نخ ها را ذخیره میکند.**

**App:شی fastapiبرای تعریف مسیرها و مدیریت درخواست ها**

**3)تعریف تابع نخ ها:**

**Myfunc:تابعی که توسط هر نخ فراخوانی میشود این تابع شماره هر نخ را به عنوان ورودی میگیرد و پیامی به لیست خروجی اضافه میکند.**

**4)تعریف مسیر API:**

**مسیر /run1/ این مسیر یک درخواست GET را پردازش می‌کند.**

**پارامتر thread\_count: تعداد نخ هایی که باید ایجاد شوند. با استفاده از Query اعتبارسنجی شده است که باید بین 1 و 10 باشد.**

**output قبل از هر اجرای جدید خالی می‌شود تا نتایج قبلی پاک شوند.**

**threads: لیستی برای نگهداری نخ های ایجاد شده.**

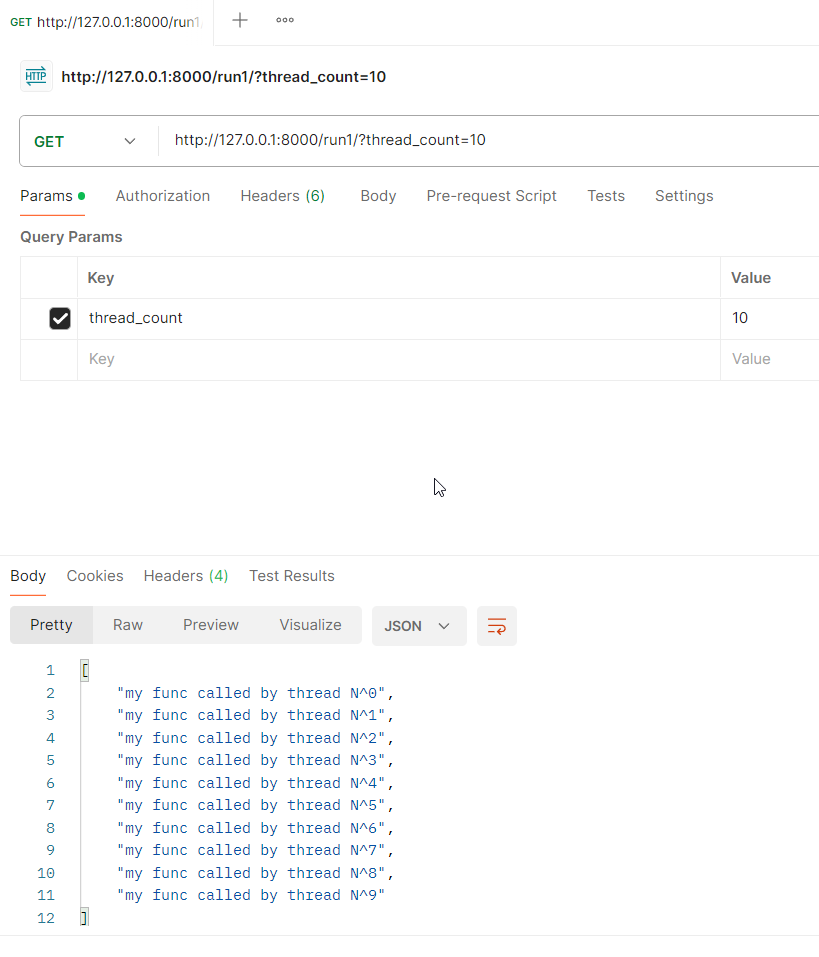
**حلقه اول: نخ ها را ایجاد و شروع می‌کند.**

**حلقه دوم: منتظر می‌ماند تا تمامی نخ ها به پایان برسند (join).**

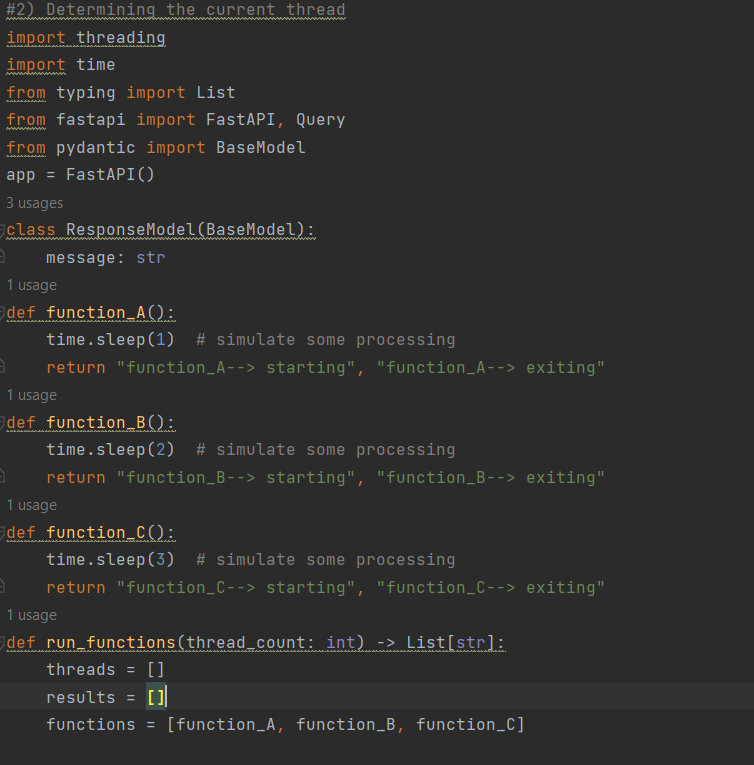
**return output: لیست output را به عنوان پاسخ برمی‌گرداند.**

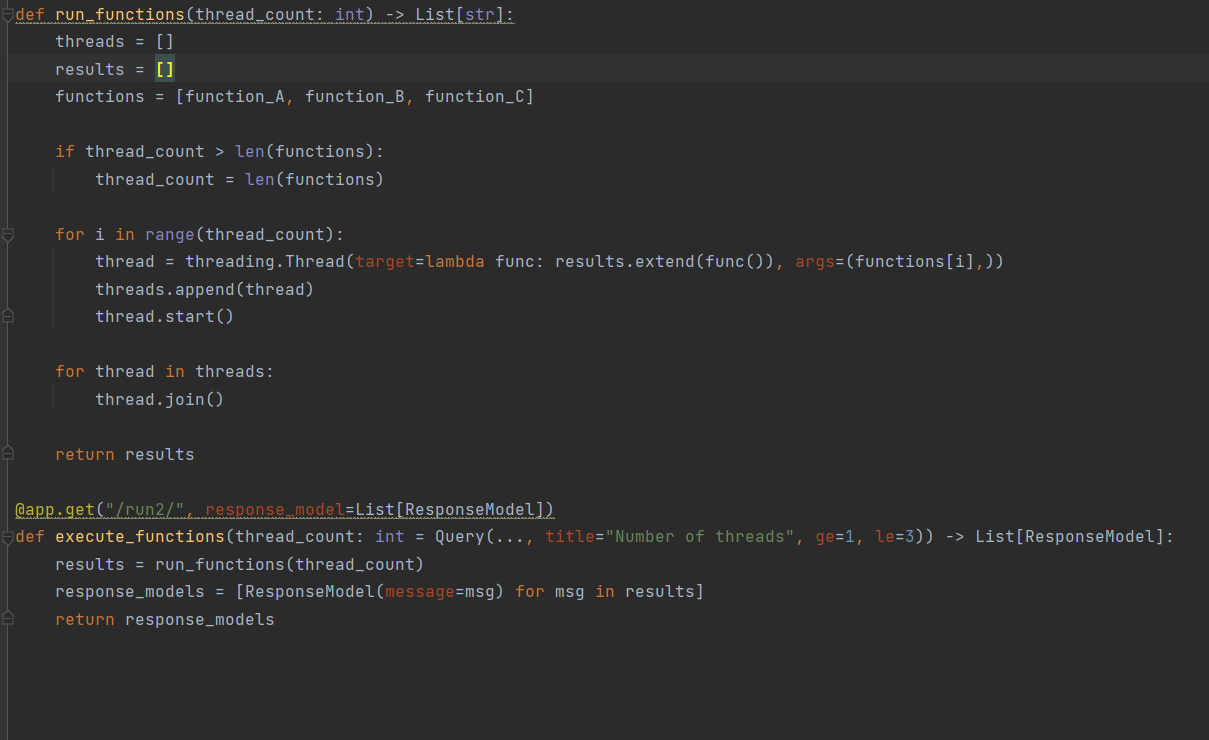
**این کد به شما امکان می‌دهد تا با فراخوانی مسیر ?thread\_count=10/run1/ و ارائه تعداد تردها به عنوان پارامتر، چندین ترد را به صورت همزمان اجرا کنید و خروجی‌های آنها را مشاهده کنید.**

**خروجی:**

****

**2) Determining the current thread**





**1)استفاده از threadingتوابع موازی**

**Threadingبرای ایجاد و مدیریت نخ های موازی استفاده میشود و تایم برای ایجاد یک تاخیر مصنوعی در هر تابع استفاده میشود.**

**2)تعریف مدل داده**

درfastapiلازم است داده از قبل تعریف شده و نوع آن مشخص شود

3)تعریف توابع موازی

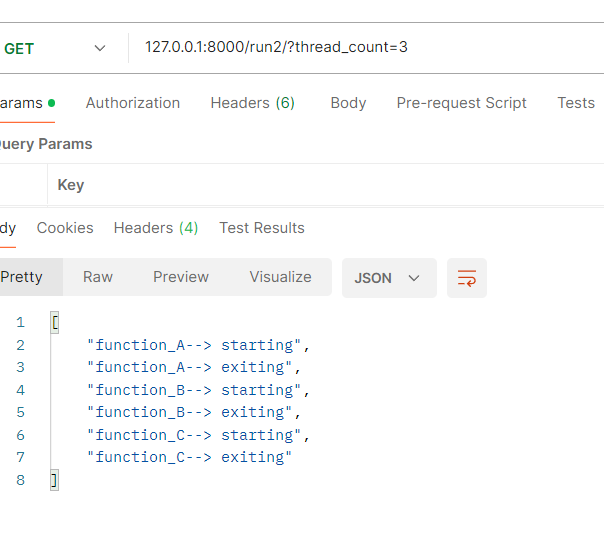
این بخش شامل تعریف سه تابع است که هر کدام یک پردازش ساده را شبیه سازی میکند هر تابع با استفاده از تایم اسلیپ یک تایر مصنوعی دادرد تا واضح مدت زمان هرکدام مشخص شود این توابع پیامی برای نشان دادن شروع و خاتمه هر تابع برمیگرداند. در این بخش، run\_functions تعداد مورد نظری از توابع موازی را به عنوان ورودی دریافت می‌کند و آن‌ها را با استفاده از threading.Thread در نخ‌های جداگانه اجرا می‌کند. پس از اتمام اجرای هر تابع، نتایج به results اضافه می‌شوند و در نهایت، تمامی نخ‌ها با استفاده از thread.join() منتظر می‌مانند تا اجرای همه توابع تمام شود

4)تعریف APIاصلی باfastapi

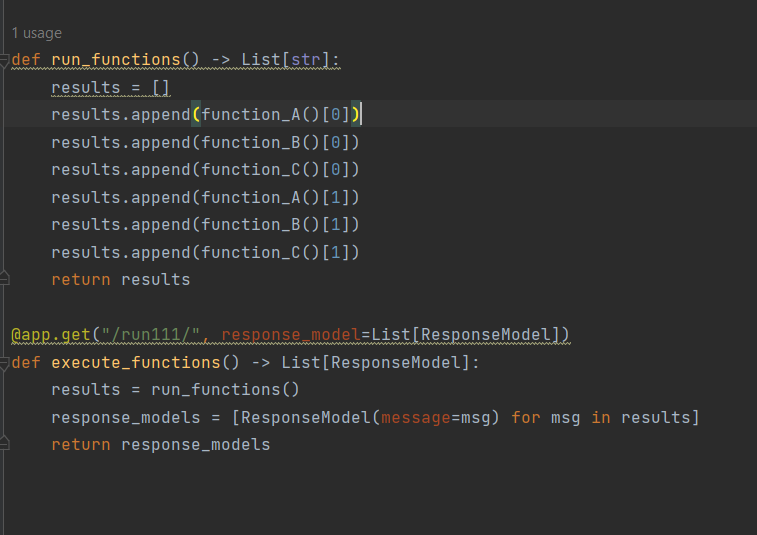
در این بخش، یک نمونه از FastAPI ساخته می‌شود و یک مسیر /run2/ برای API تعریف می‌شود. این API یک پارامتر thread\_count را از کاربر دریافت می‌کند که تعداد نخ‌هایی که باید برای اجرای توابع موازی استفاده شود را مشخص می‌کند. در پایان، نتایج به صورت لیستی از ResponseModel تبدیل شده و به عنوان خروجی API ارسال می‌شوند.

این توضیحات به شما کمک می‌کنند تا هر بخش از کد را درک کنید و بفهمید که چگونه توابع موازی با استفاده از threading در یک برنامه FastAPI پیاده‌سازی شده‌اند.

خروجی:





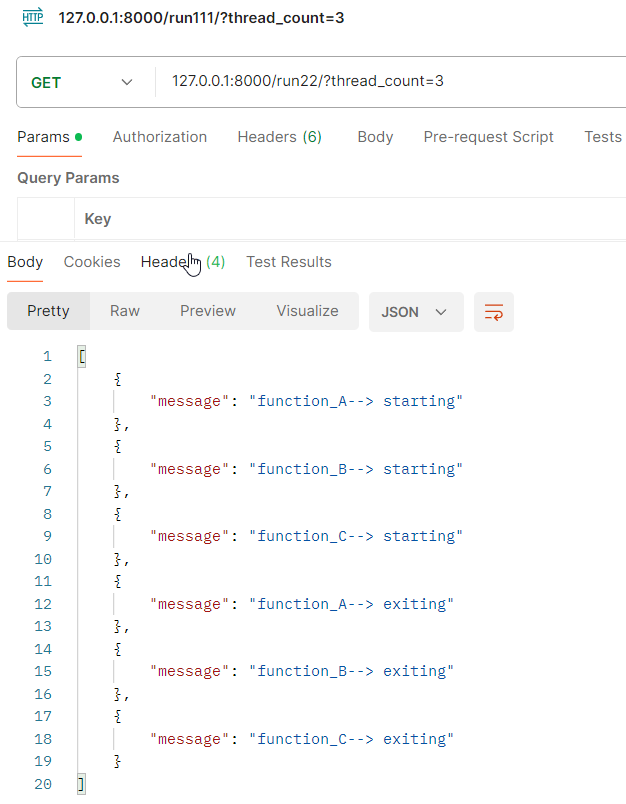


در این بخش از کد، توابع function\_A، function\_B و function\_C صدا زده می‌شوند و خروجی آن‌ها به ترتیب به لیست results اضافه می‌شود. این توابع به ترتیب دو عنصر در لیست برمی‌گردانند:

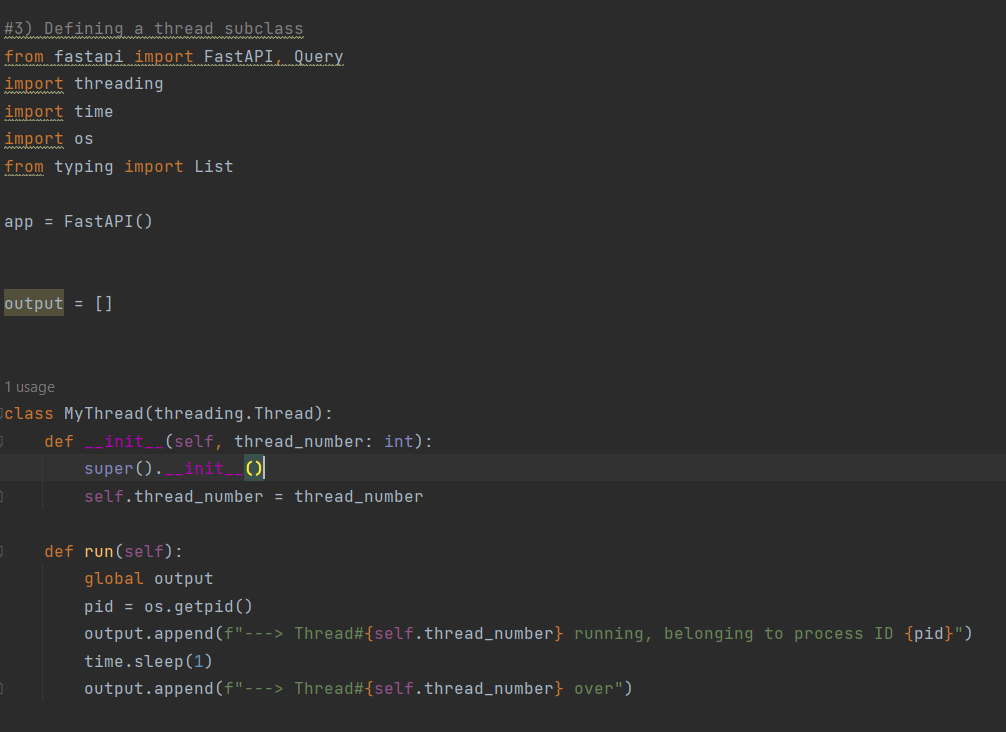
1. function\_A()[0], function\_B()[0], function\_C()[0]: این بخش از کد، پیام شروع از هر یک از توابع function\_A، function\_B و function\_C را به لیست results اضافه می‌کند. به عبارت دیگر، اولین عنصر بازگشتی هر یک از این توابع که نماینده‌ی پیام شروع است به لیست اضافه می‌شود.
2. function\_A()[1], function\_B()[1], function\_C()[1]: این بخش از کد، پیام خاتمه از هر یک از توابع function\_A، function\_B و function\_C را به لیست results اضافه می‌کند. به عبارت دیگر، دومین عنصر بازگشتی هر یک از این توابع که نماینده‌ی پیام خاتمه است به لیست اضافه می‌شود.

بنابراین، با این کد تضمین می‌شود که پیام‌های شروع و خاتمه از هر یک از توابع function\_A، function\_B و function\_C به ترتیب صحیح به لیست results اضافه و برگشت داده شوند.

خروجی:



**3) Defining a thread subclass**





توضیح کد:

وارد کردن کتابخانه هایfastapi,threading, time,os,list

و ایجاد یک شی از FastAPI و تعریف یک متغیر سراسریoutput

تعریف کلاس My threads,و **\_\_init\_\_ method**: این متد سازنده کلاس است که شماره رشته (thread number را به عنوان ورودی می‌پذیرد و آن را به یک متغیر نمونه اختصاص می‌دهد

.run method این متد زمانی که رشته شروع به اجرا می‌کند، فراخوانی می‌شود. در این متد:

شناسه پردازش فعلی با استفاده از os.getpid() به دست می‌آید.

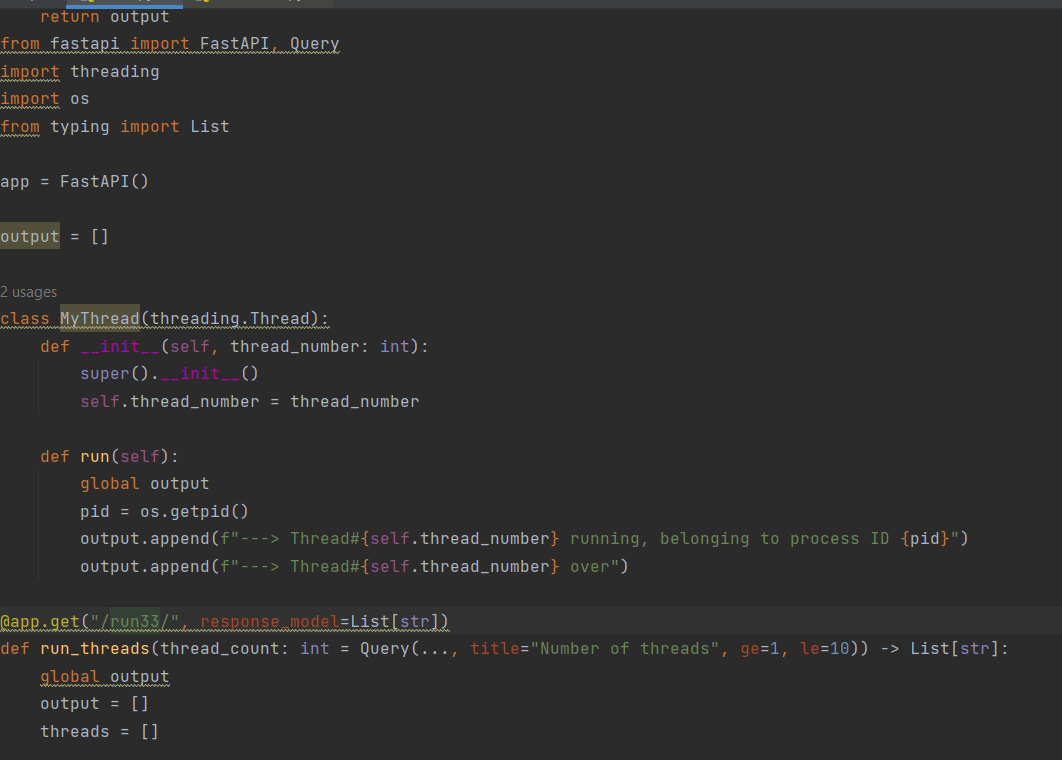
پیامی به متغیر output اضافه می‌شود که نشان می‌دهد رشته در حال اجرا است

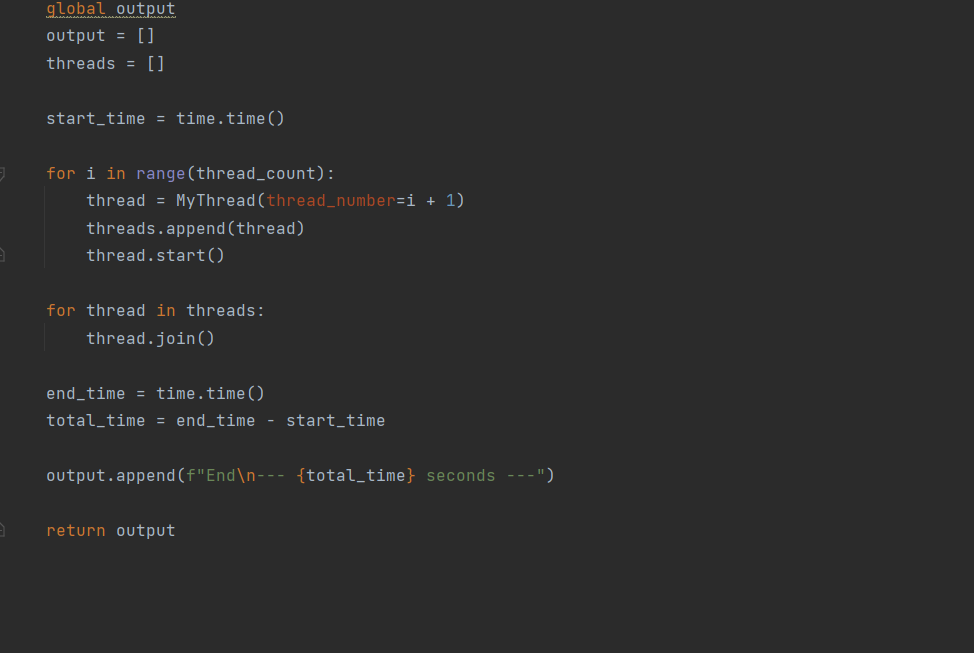
در انهایت با ارث بری از کلاس اولیه و ایجاد یک زیر کلاس به مقدار 10 عدد نخ را همزمان اجرا میکند و با شماره آنها به خروجی میبرد در ابتدا آغاز را نمایش داده و نخ ها یکی یکی پایان میابد و در نهایت مدت زمان اجرای کل را به نمایش میگذارد

خروجی:

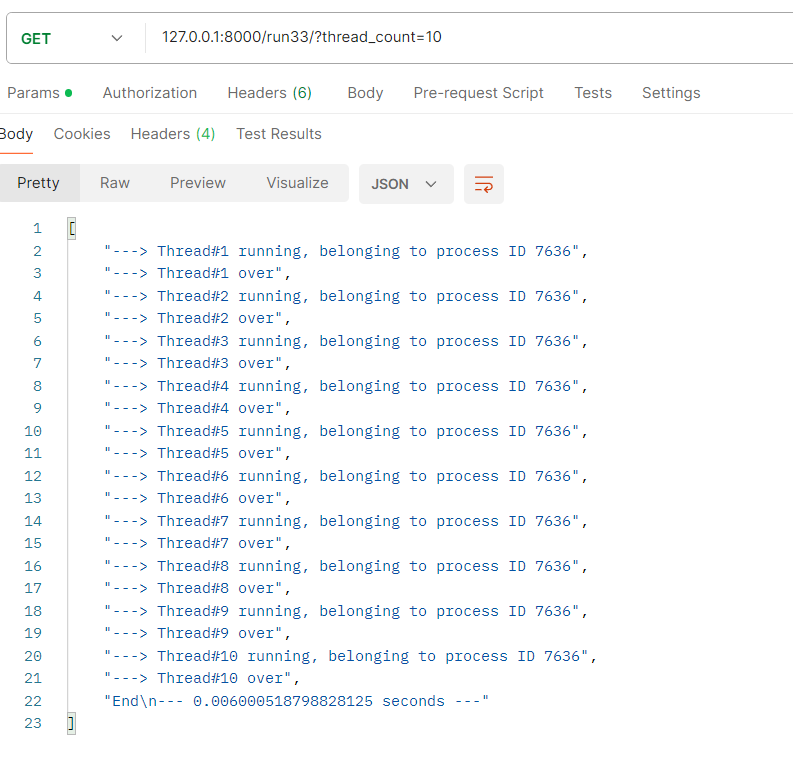


در سناریو بعدی تفاوتی که حاصل شده با حذف تایم اسلیپ انجام شده است که با حذف آن هرنخ ابتدا اجرا شده سپس اتمام میابد و تا اتمام نیافته نخ بعدی اجرا نمیشود

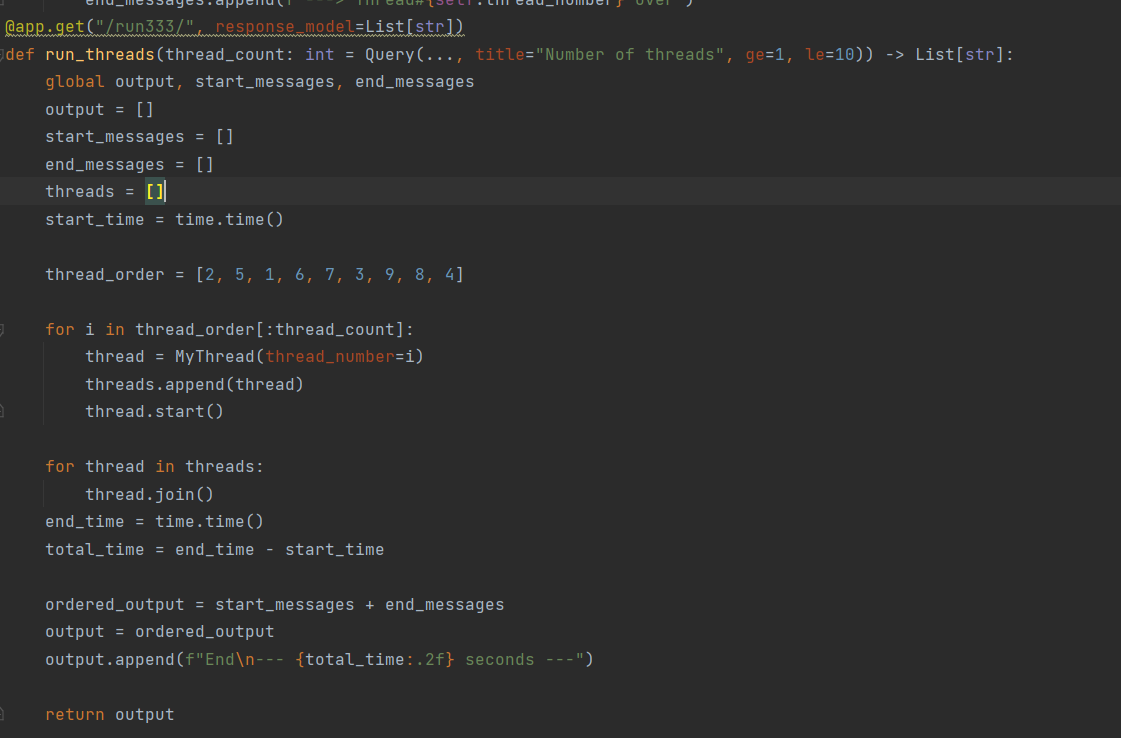




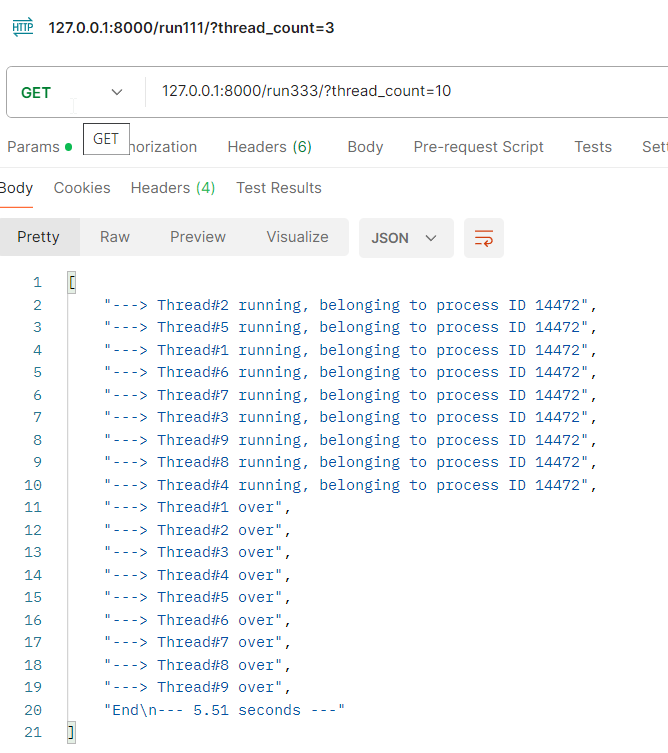
خروجی:

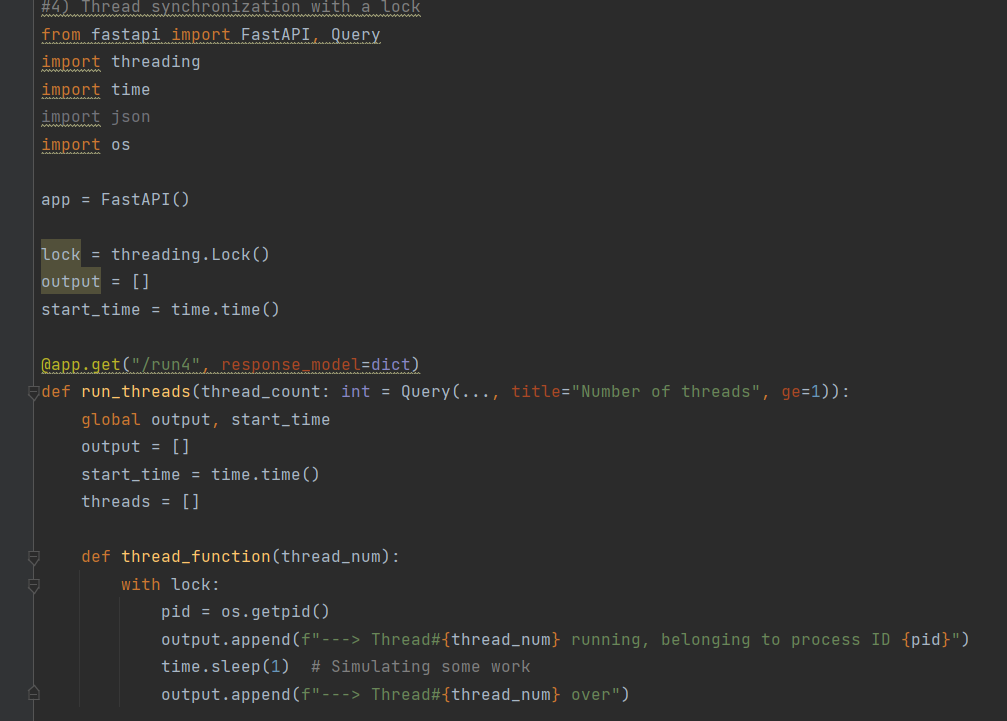
در سناریو آخر یک لیست قرار داده شده که به ترتیب آن نخ با همان شماره اجرا میشود

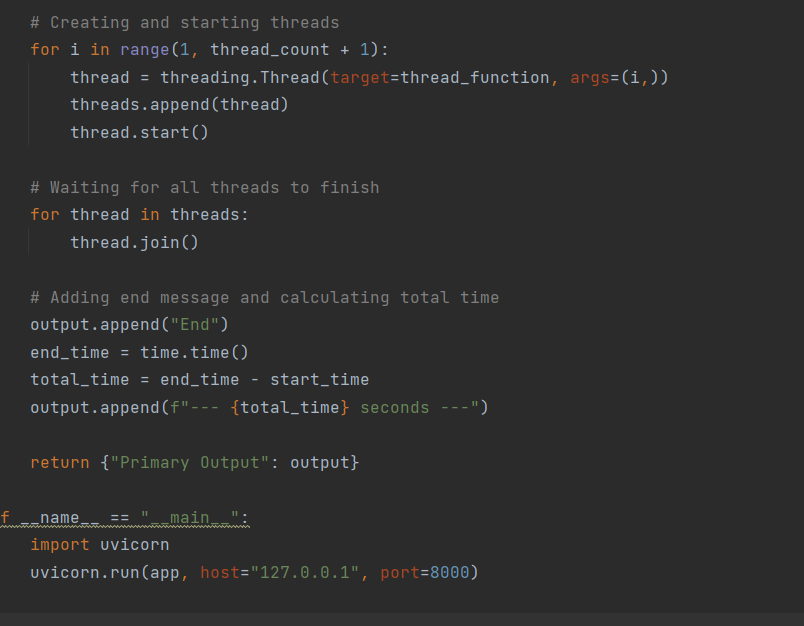




که خروجی آن بصورت:

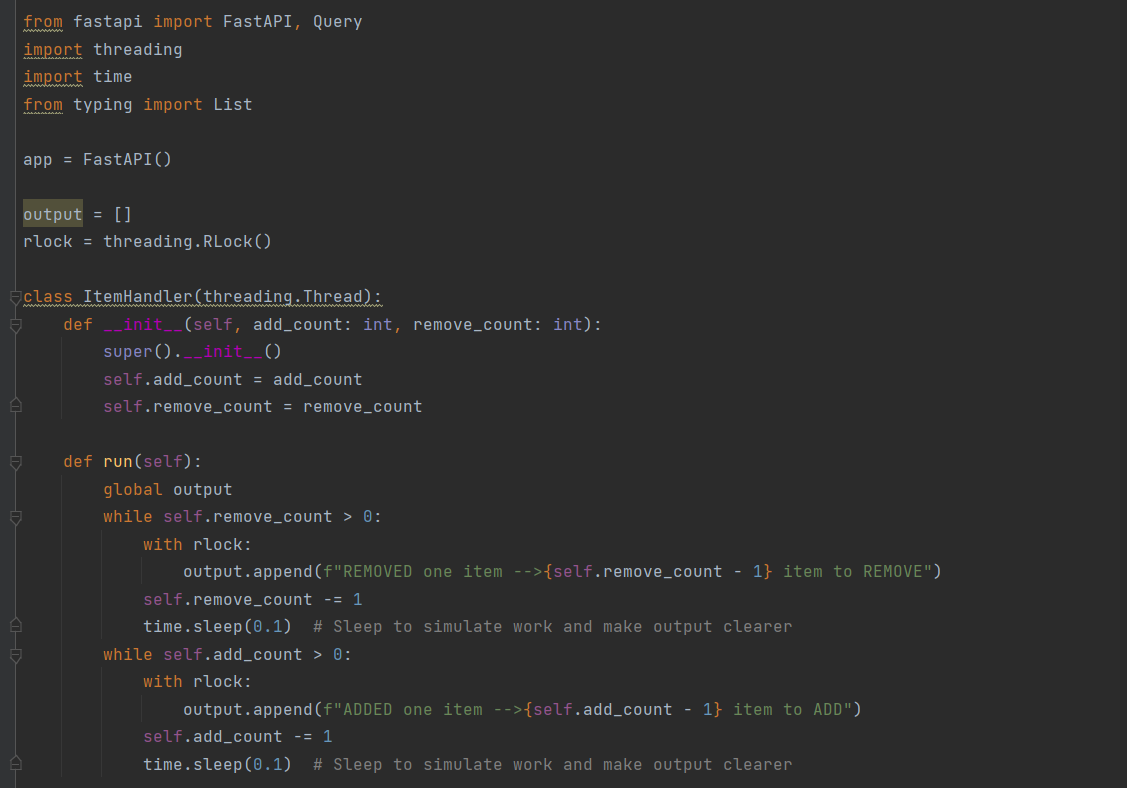


**4) Thread synchronization with a lock**



تابعی برای اجرای هر نخ قرار داده شده که با استفاده از قفل لاک به محدود کردن دسترس میپردازد و صبر میکند تا تمامی نخ ها به اتمام برسد سپس کار با قفل تمام میشود.پس هرنخ شروع به اجرا کرده . تمام میشود

**5) Thread synchronization with RLock**





این کد یک سرویس وب API با استفاده از کتابخانه FastAPI ایجاد می‌کند که قابلیت افزودن و حذف موارد (items) را به صورت همزمان و در چندین نخ فراهم می‌کند. این سرویس از طریق یک مسیر GET به نام /run5/ قابل دسترسی است و دو پارامتر ورودی (add\_count و remove\_count) دریافت می‌کند که تعداد مواردی را که باید اضافه یا حذف شوند مشخص می‌کندoutput. یک لیست جهانی برای ذخیره پیام‌های تولید شده توسط رشته‌ها.

rlock یک قفل بازگشتی (reentrant lock) که برای اطمینان از دسترسی همزمان ایمن به output استفاده می‌شود.که همانطور که میدانیم میتواند چندین بار اتخاذ شود. \_\_init\_\_ method متد سازنده که تعداد مواردی که باید اضافه و حذف شوند را به عنوان ورودی می‌پذیرد و آن‌ها را به متغیرهای نمونه اختصاص می‌دهد.

runmethod این متد زمانی که نخ شروع به اجرا می‌کند، فراخوانی می‌شود. این متد دو حلقه دارد: اولین حلقه برای حذف موارد است و تا زمانی که remove\_count به صفر برسد، ادامه می‌یابد. در هر تکرار، پیام مربوط به حذف یک مورد به output اضافه می‌شود و remove\_count کاهش می‌یابد.دومین حلقه برای افزودن موارد است و تا زمانی که add\_count به صفر برسد، ادامه می‌یابد. در هر تکرار، پیام مربوط به افزودن یک مورد به output اضافه می‌شود و add\_count کاهش می‌یابد.time.sleep: برای شبیه‌سازی عملیات و ایجاد تاخیر در هر تکرار حلقه استفاده می‌شود تا خروجی قابل مشاهده‌تر شود.

handle\_items function: این تابع زمانی که مسیر /run5/ درخواست می‌شود، فراخوانی می‌شود.

پارامتر add\_count: تعداد مواردی که باید اضافه شوند. از طریق Query parameter دریافت می‌شود و باید بزرگتر یا مساوی صفر باشد.

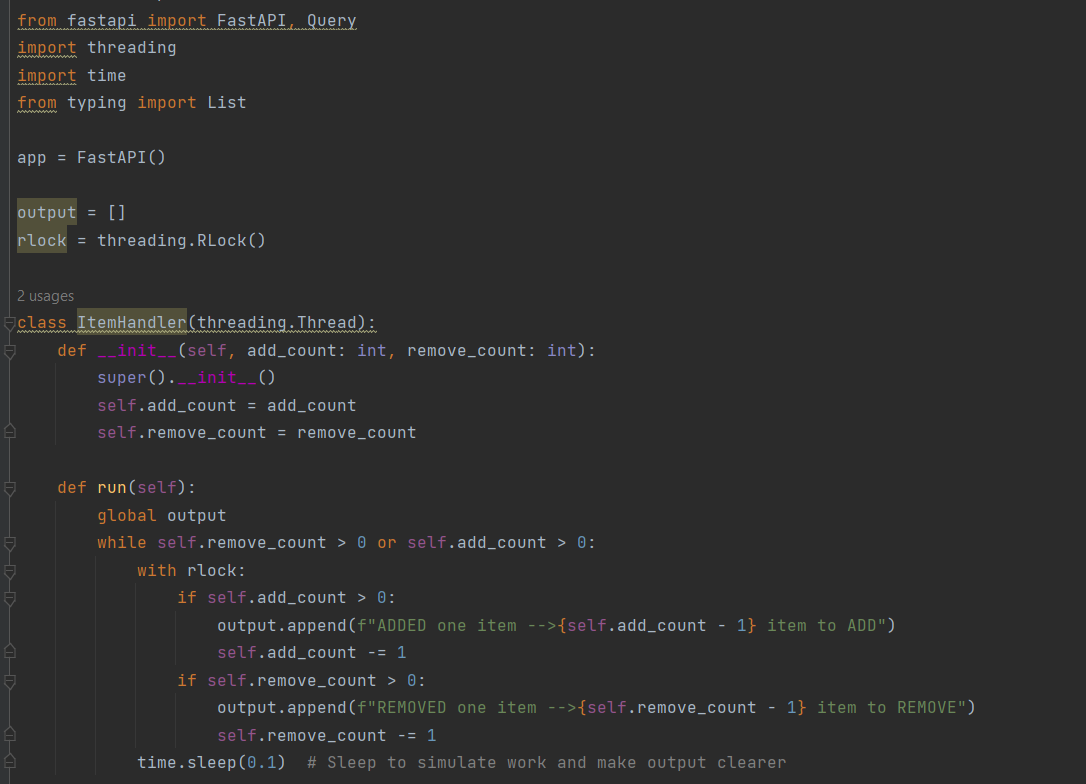
پارامتر remove\_count: تعداد مواردی که باید حذف شوند. از طریق Query parameter دریافت می‌شود و باید بزرگتر یا مساوی صفر باشد.

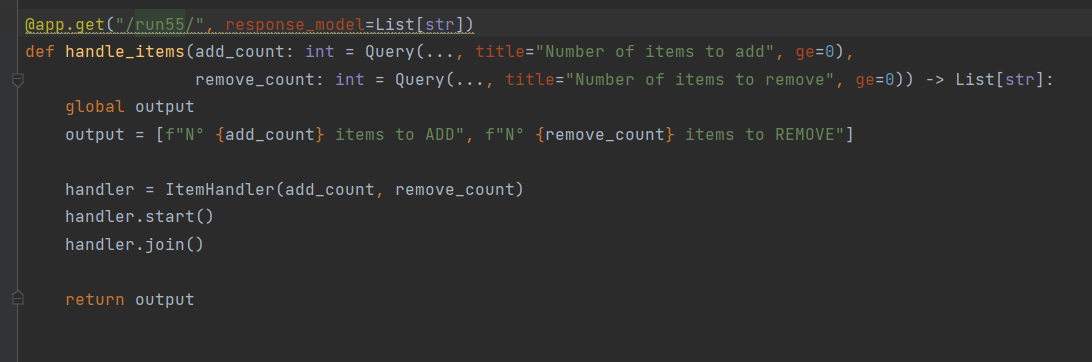
تنظیم اولیه output: لیست output با پیام‌هایی که تعداد مواردی که باید اضافه و حذف شوند را نشان می‌دهد، مقداردهی می‌شود.

ایجاد و شروع رشته ItemHandler: یک رشته جدید از کلاس ItemHandler با مقادیر add\_count و remove\_count ایجاد می‌شود و شروع به کار می‌کند.

انتظار برای اتمام رشته: برنامه منتظر می‌ماند تا رشته به پایان برسد.

بازگشت output: لیست output که شامل تمام پیام‌ها است، به عنوان پاسخ API بازگردانده می‌شود.

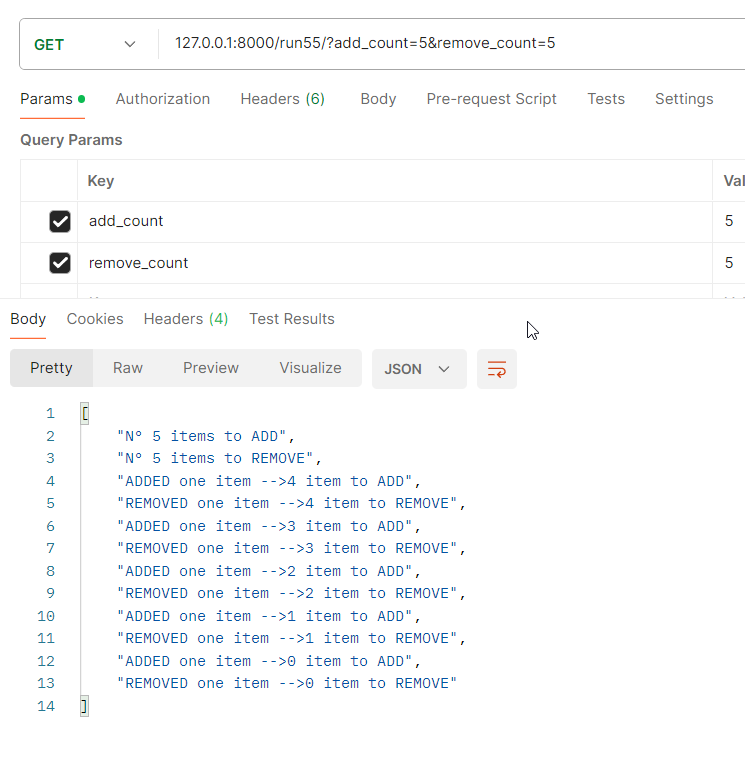




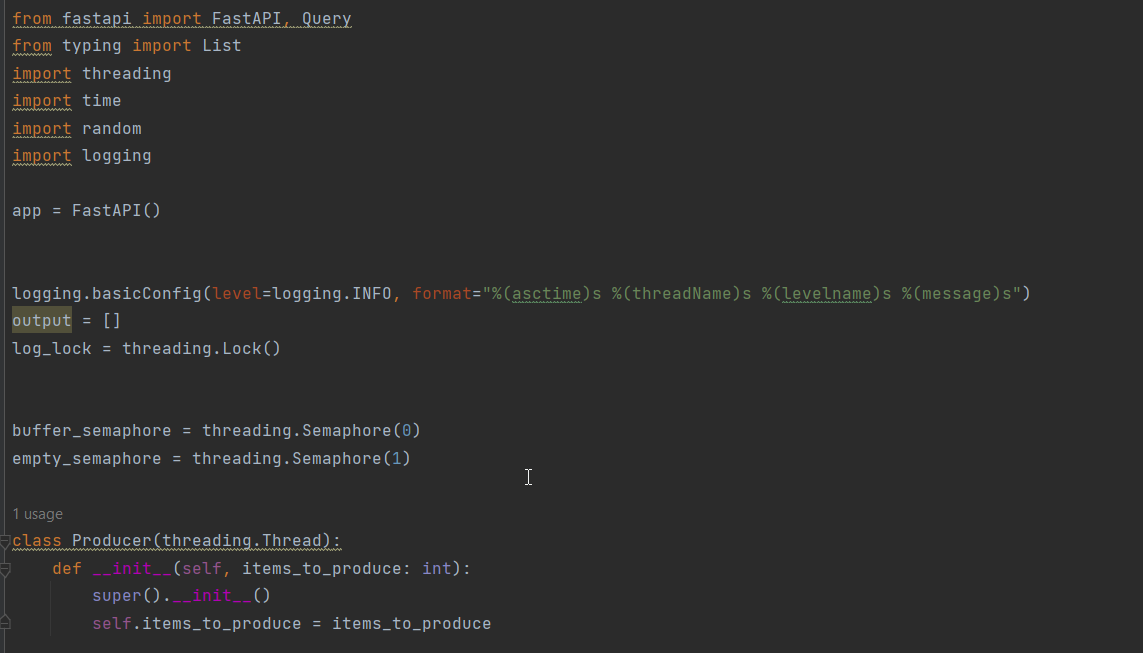
الگوریتم حذف و اضافه را به صورت متناوب انجام دهیم یا ترتیب انجام آن‌ها را تغییر دهیم. به عنوان مثال، ابتدا یک مورد اضافه کنیم، سپس یک مورد حذف کنیم، و به همین ترتیب ادامه دهیم. همچنین می‌توانیم خروجی را به ترتیب متفاوتی در لیست قرار دهیم.

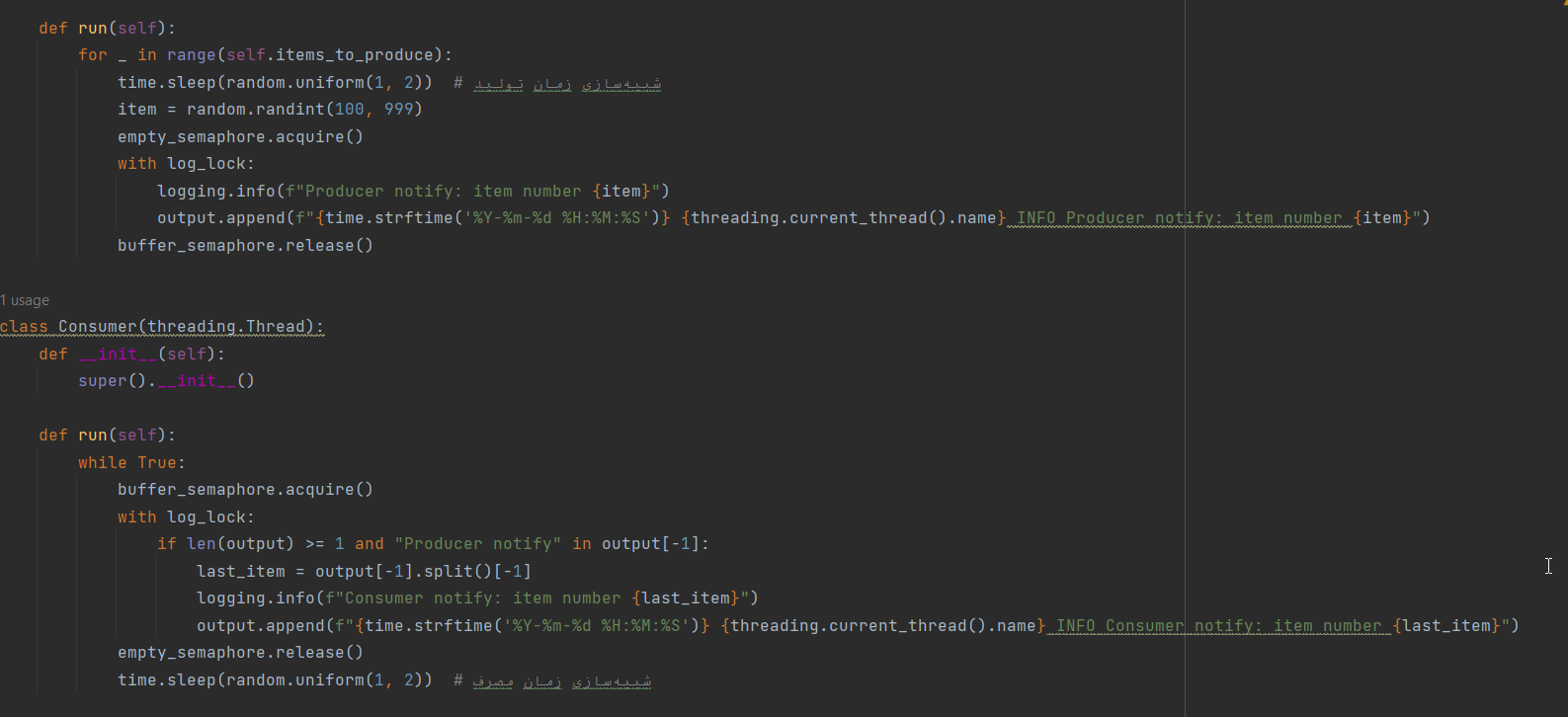
**تغییر در الگوریتم** run در متد run، همزمان حذف و اضافه به صورت متناوب انجام می‌شود تا ترتیب خروجی تغییر کند.از حلقه while برای مدیریت همزمان اضافه و حذف استفاده می‌شود تا هر بار که یکی از عملیات‌ها انجام می‌شود، عملیات دیگر بررسی و اجرا شود (در صورت موجود بودن).این مسیر مانند قبل، تعداد مواردی که باید اضافه و حذف شوند را از طریق پارامترهای query دریافت می‌کند .خروجی اولیه به صورت یک لیست از پیام‌های شروع اضافه و حذف تنظیم می‌شود.یک شیء از کلاس ItemHandler ایجاد و اجرا می‌شود و سپس منتظر می‌ماند تا عملیات به پایان برسد.وقتی این API فراخوانی شود، خروجی شامل پیام‌های افزودن و حذف به صورت متناوب خواهد بود

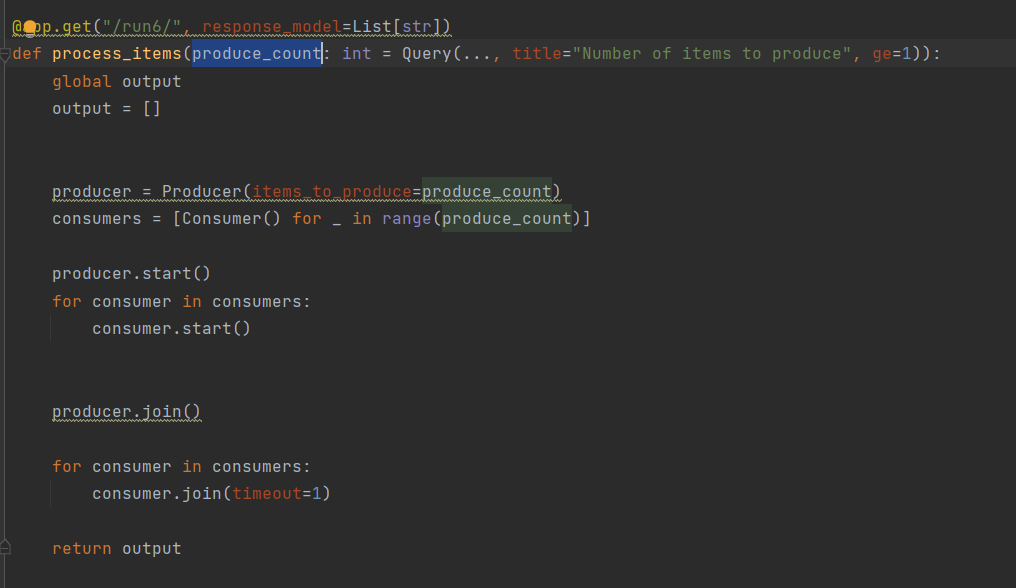
خروجی:



**6) Thread synchronization with semaphores**







این کد یک API ساده با استفاده از FastAPI و threading ایجاد می‌کند که تولید و مصرف آیتم‌ها را شبیه‌سازی می‌کند. این فرآیند با استفاده از کلاس‌های Producer و Consumer انجام می‌شود که هر کدام به عنوان یک نخ اجرا می‌شوند. buffer\_semaphore برای کنترل تعداد آیتم‌های موجود در بافر استفاده می‌شود.

empty\_semaphore برای کنترل دسترسی به بافر در زمان خالی بودن استفاده می‌شود.این کلاس یک نخ است که آیتم‌هایی تولید می‌کند.

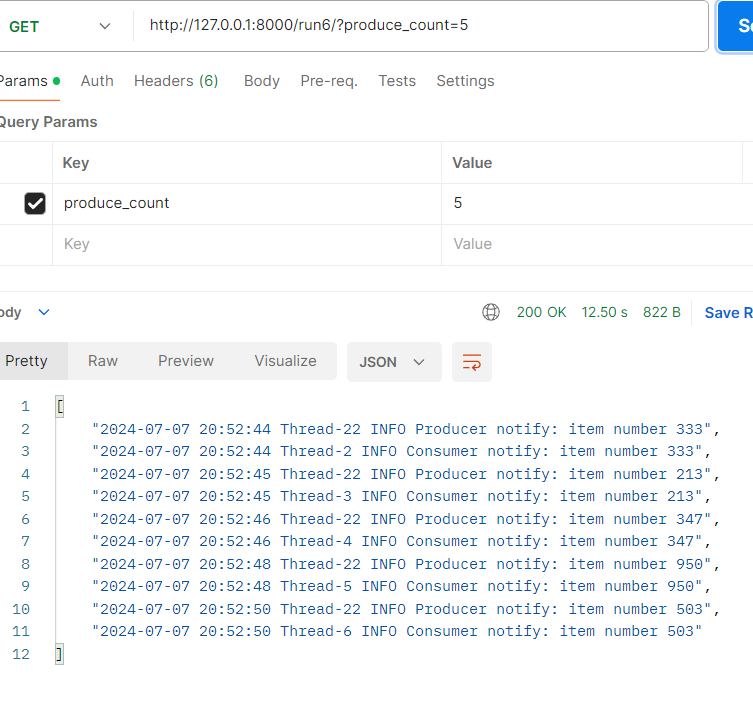
\_\_init\_\_سازنده کلاس است که تعداد آیتم‌هایی که باید تولید شوند را دریافت می‌کند.

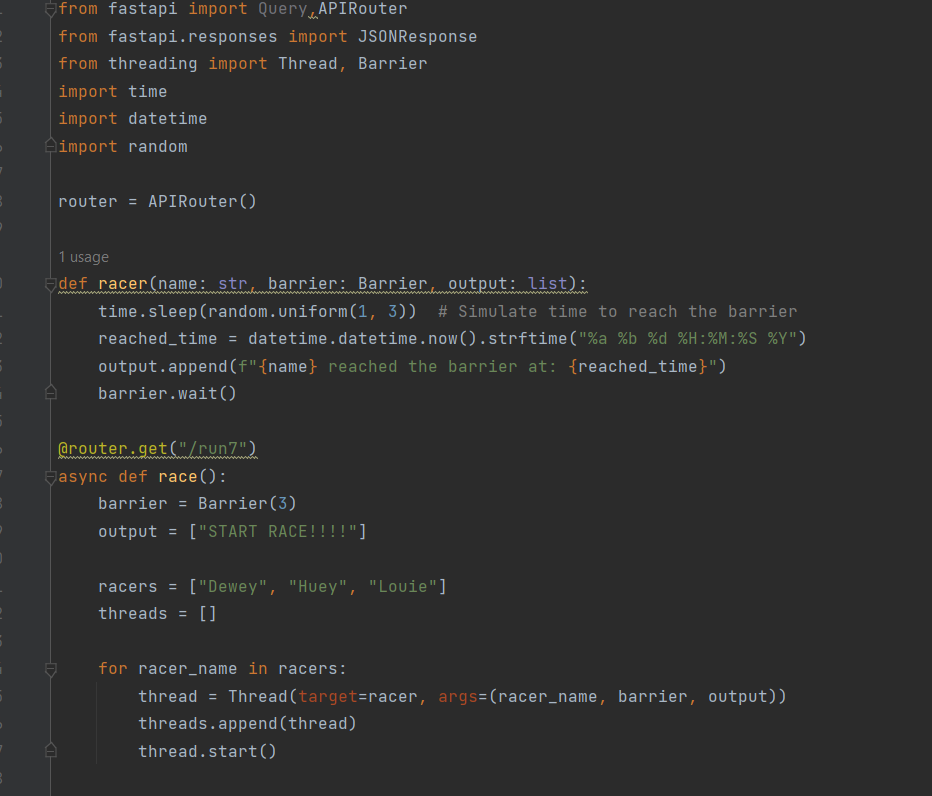
run متدی است که توسط نخ اجرا می‌شود و در آن آیتم‌ها تولید می‌شوند. هر آیتم با یک تاخیر زمانی تصادفی تولید می‌شود و در بافر قرار می‌گیرد. پیام‌های مربوط به تولید آیتم‌ها در output ذخیره می‌شوند

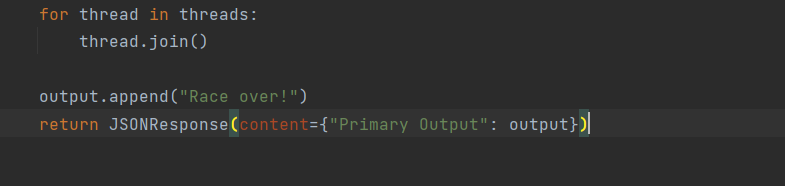
تعریف کلاس:consumerاین کلاس یک نخ است که آیتم‌ها را مصرف می‌کند. \_\_init\_\_سازنده کلاس است.

run متدی است که توسط نخ اجرا می‌شود و در آن آیتم‌ها از بافر برداشته می‌شوند. پیام‌های مربوط به مصرف آیتم‌ها در ذخیره output می‌شوند. نخ به طور مداوم کار می‌کند تا زمانی که آیتمی برای مصرف وجود نداشته باشد.

produce\_count تعداد آیتم‌هایی است که باید تولید شوند و به عنوان پارامتر Query دریافت می‌شود. output با یک لیست خالی بازنشانی می‌شود.یک شیء از کلاس Producer و تعدادی شیء از کلاس Consumer بر اساس تعداد آیتم‌ها ایجاد می‌شود.نخ‌های تولید و مصرف شروع به کار می‌کنند و با join منتظر می‌مانند تا عملیات به پایان برسد.خروجی نهایی برگردانده می‌شود.

 **7)Thread synchronization with a barrier**

****



در اینجا با تعریف تابع racer یک شبیه سازی از رسیدن مسابقه دهنده به مانع است و barrierمانعی که نخ ها باید منتظر بمانند تا همه به آن برسند و outputلیستی که زمان رسیدن مسابقه دهنده به مانع را ذخیره میکندو در نهایت output.append(...پیام رسیدن به مانع رابه لیست خروجی اضافه میکندوbarrier.wait()نخ را تا رسیدن همه نخ ها به مانع متوقف میکند

این مسیر یک مسابقه نخ‌ها را شبیه‌سازی می‌کند و خروجی را به صورت JSON بازمی‌گرداند.

1) barrier = Barrier(3)ایجاد یک مانع برای همگام‌سازی سه نخ.

2) output = ["START RACE!!!!"]لیستی برای ذخیره پیام‌های مسابقه.

3) racers = ["Dewey", "Huey", "Louie"]: لیست مسابقه‌دهنده‌ها.

4) threads = []: لیستی برای ذخیره نخ‌ها.

. for racer\_name in racers برای هر مسابقه‌دهنده:

thread = Thread(target=racer, args=(racer\_name, barrier

. output)): ایجاد یک نخ جدید برای هر مسابقه‌دهنده که تابع racer را اجرا می‌کند.

. threads.append(thread)اضافه کردن نخ به لیست threads.

:. thread.start(): شروع نخ.

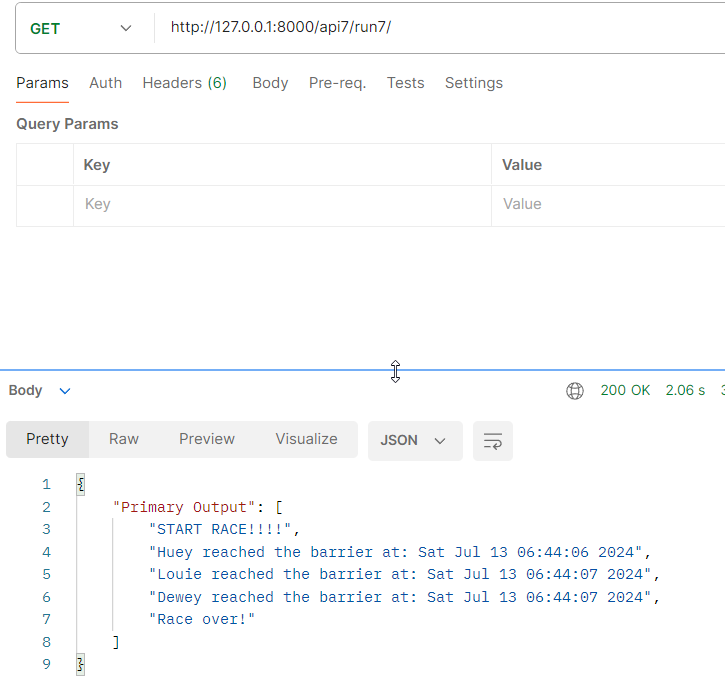
. for thread in threads:برای هر نخ:

. thread.join():منتظر بمان تا نخ پایان یابد.

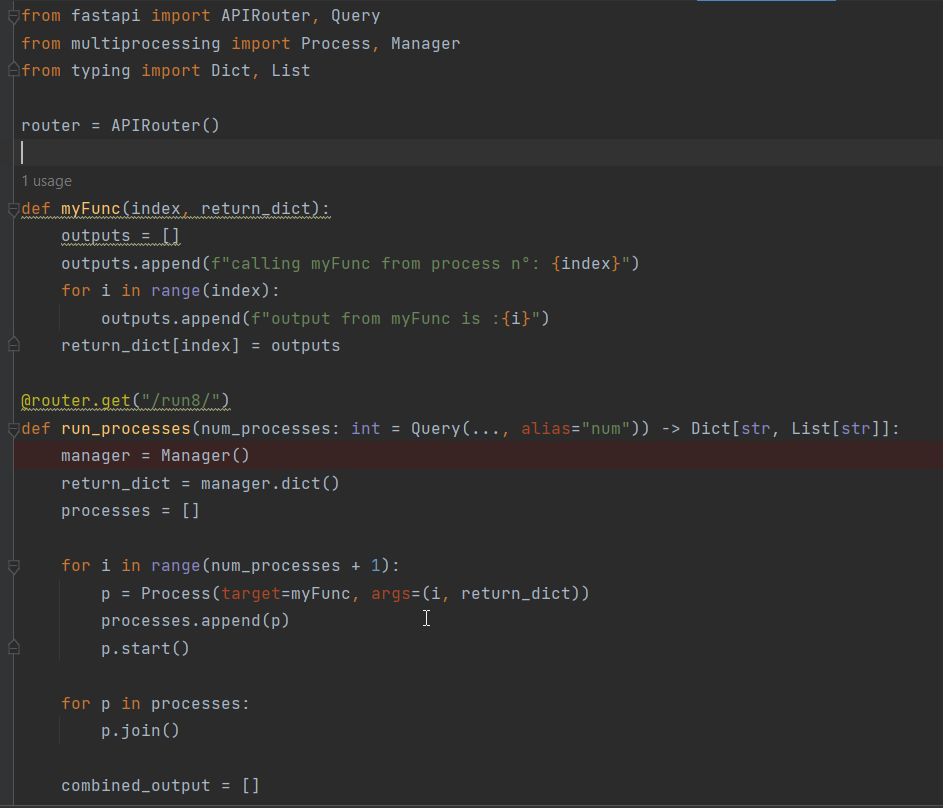
. output.append("Race over!"): اضافه کردن پیام پایان مسابقه به لیست output.

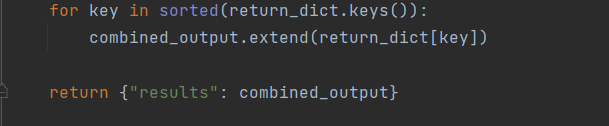
. return JSONResponse(content={"Primary Output": output}): بازگرداندن خروجی به صورت JSON.

این کد با استفاده از FastAPI یک مسابقه نخ‌ها را شبیه‌سازی می‌کند که هر نخ به صورت تصادفی پس از تأخیر به مانع می‌رسد و سپس منتظر می‌ماند تا همه نخ‌ها به مانع برسند. پس از اتمام مسابقه، نتیجه به صورت JSON به کلاینت بازگردانده می‌شود.

**خروجی:**

**Spawning a process**



****

**تعریف تابع myFunc**:

این تابع برای هر پردازش فراخوانی می‌شود و خروجی‌ها را تولید و در return\_dict ذخیره می‌کند.

. Index شماره پردازش.

. return\_dict:دیکشنری مشترکی که برای ذخیره خروجی‌ها بین پردازش‌ها استفاده می‌شود.

. outputs: لیستی برای ذخیره خروجی‌های تولید شده در هر پردازش.

. {index}"): outputs.append(f"calling myFunc from.

. process n°افزودن پیام شروع پردازش به لیست outputs.

. for i in range(index):حلقه‌ای برای تولید خروجی‌ها بر اساس شماره پردازش.

. return\_dict[index] = outputs: ذخیره لیست outputs در return\_dict با کلید index.

این مسیر تعداد پردازش‌ها را به عنوان ورودی دریافت می‌کند.

num\_processes: int = Query(..., alias="num"): ودریافت تعداد پردازش‌ها به عنوان پارامتر query با نام num.

manager = Manager(): ایجاد یک Manager برای مدیریت داده‌های مشترک بین پردازش‌ها.

return\_dict = manager.dict(): ایجاد یک دیکشنری مشترک برای ذخیره خروجی‌ها.

processes = []: لیستی برای ذخیره پردازش‌ها.

for i in range(num\_processes + 1): حلقه‌ای برای ایجاد و شروع پردازش‌ها.

p = Process(target=myFunc, args=(i, return\_dict)):ایجاد یک پردازش جدید که تابع myFunc را با آرگومان‌های i و return\_dict اجرا می‌کند.

processes.append(p): اضافه کردن پردازش به لیست processes.

p.start(): شروع پردازش.

for p in processes: حلقه‌ای برای اطمینان از اینکه همه پردازش‌ها به پایان رسیده‌اند.

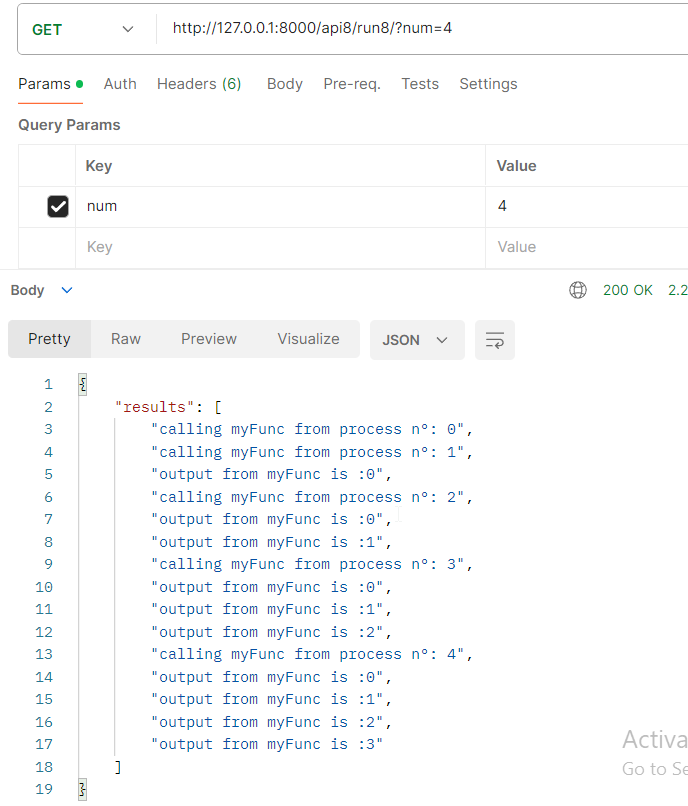
. p.join():منتظر ماندن برای پایان یافتن پردازش.

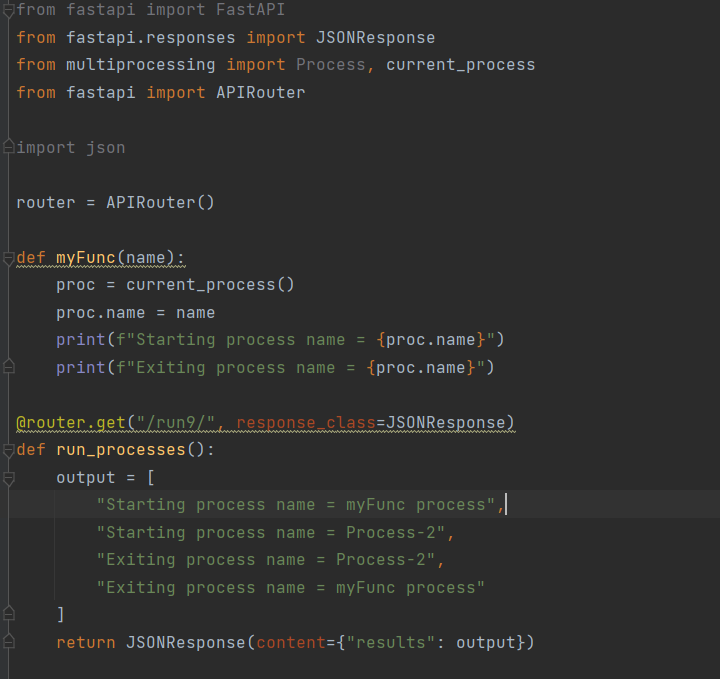
. combined\_output = []:لیستی برای ترکیب خروجی‌های پردازش‌ها.

for key in sorted(return\_dict.keys()): حلقه‌ای برای مرتب‌سازی و ترکیب خروجی‌ها.

. combined\_output.extend(return\_dict[key]):افزودن خروجی‌های هر پردازش به لیست combined\_output.

return {"results": combined\_output}: بازگرداندن خروجی ترکیبی به صورت JSON.

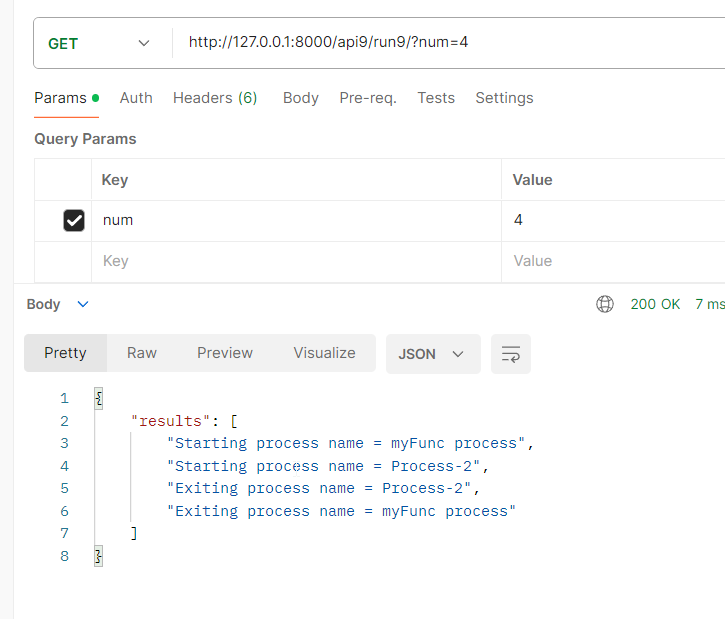


**Naming a process**

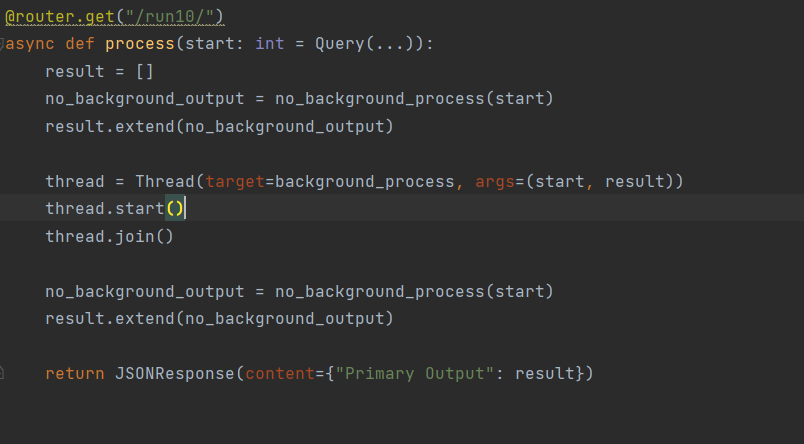
**تعریف تابع myFunc**

* این تابع برای هر فرآیند فراخوانی می‌شود و نام فرآیند را تنظیم و چاپ می‌کند.
* name نامی که برای فرآیند تنظیم می‌شود.
* proc = current\_process()دریافت فرآیند جاری.
* proc.name = name تنظیم نام فرآیند.
* print(f"Starting process name = {proc.name}") چاپ پیام شروع فرآیند.
* print(f"Exiting process name = {proc.name}") چاپ پیام خروج از فرآیند

خروجی:



**Running processes in the background**



در کد بالا با استفاده از تابع no\_BGو مقدار پارامتر نشان میدهد که کدام رشته و با چه اندیسی مقدار باز گردانده شده توسط تابع در خروجی نمایان نمی شود و در پس زمینه انجام میشود deamon=True

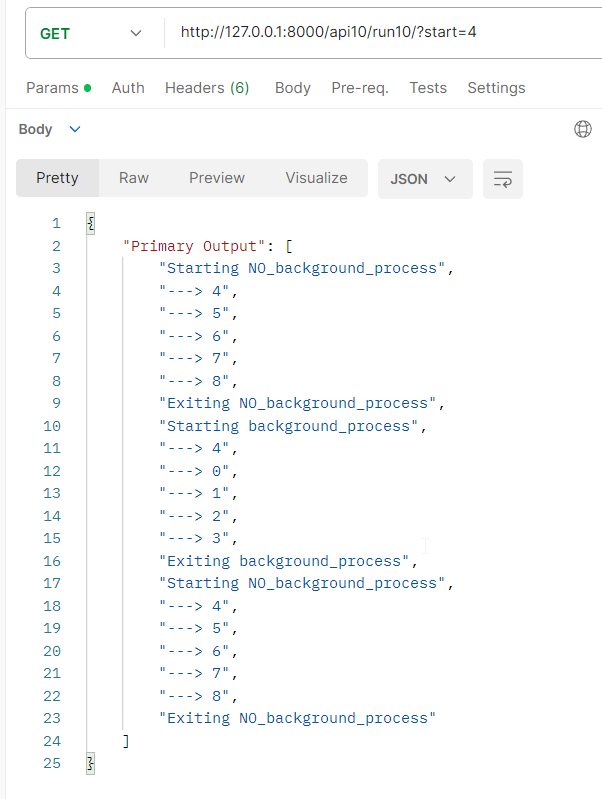
اما در تابع BGبدین صورت عمل میشود که هر چه در این تابع دیده میشود بازتاب آن قابل نمایش است

no\_background\_process(start: int): این تابع یک لیست از رشته‌ها را ایجاد می‌کند که شامل پیام‌هایی است که به صورت سریالی (بدون همزمانی) ایجاد می‌شوند. این پیام‌ها به عنوان شروع و خاتمه‌ی فرآیند بدون پس‌زمینه (no\_background\_process) نمایش داده می‌شود

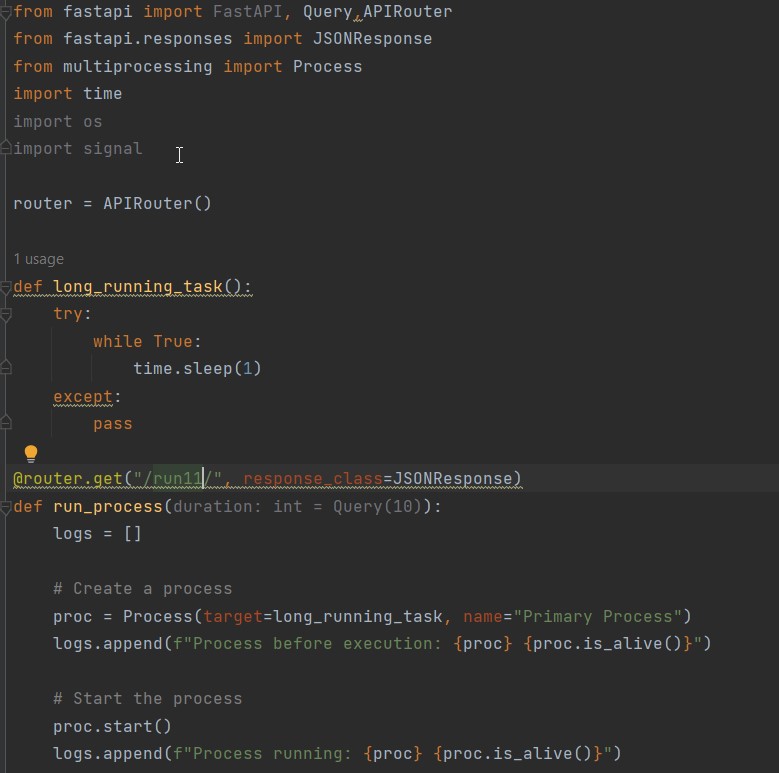
background\_process(start: int, result: list): این تابع یک لیست از رشته‌ها را به عنوان ورودی دریافت می‌کند و پیام‌هایی را ایجاد می‌کند که نشان دهنده شروع و خاتمه‌ی فرآیند پس‌زمینه (background\_process) است. این پیام‌ها به لیست نتیجه در اضافه نمایش داده می‌شوند.

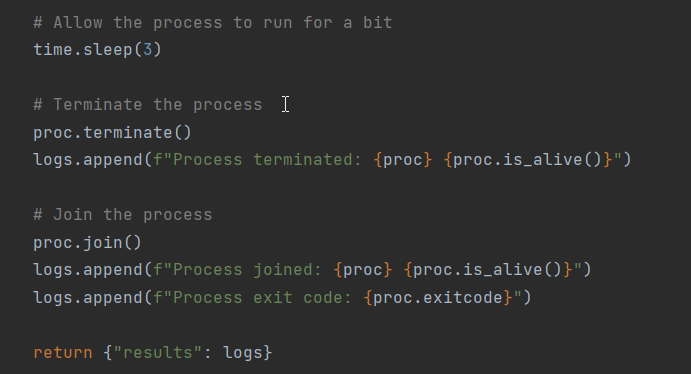
process(start: int = Query(...)): این تابع مسئول ایجاد و مدیریت فرآیندها و نخ‌ها برای ایجاد خروجی نهایی است. ابتدا تابع no\_background\_process برای ایجاد خروجی بدون پس‌زمینه فراخوانی می‌شود و نتایج آن به لیست نتیجه اضافه می‌شود.

در این قسمت از کد، یک فرآیند پس‌زمینه (background\_process) با استفاده از کلاس Process از multiprocessing ایجاد می‌شود. این فرآیند با استفاده از پارامتر daemon=True در زمان ایجاد، به عنوان یک فرآیند daemon تعریف می‌شود. این بدان معنی است که اگر برنامه اصلی به پایان برسد، فرآیند پس‌زمینه همچنان ادامه می‌یابد. تابع background\_process در این فرآیند اجرا می‌شود و نتایج آن به لیست نتیجه اضافه می‌شود.



**Killing a process:**





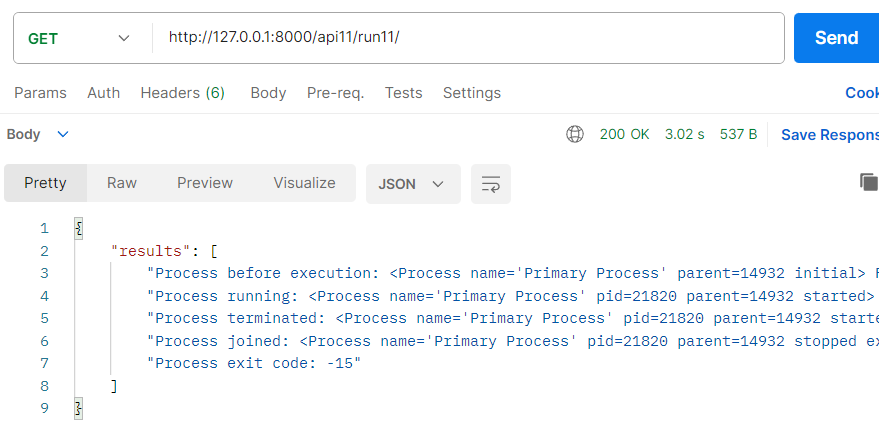
در کد بالا :

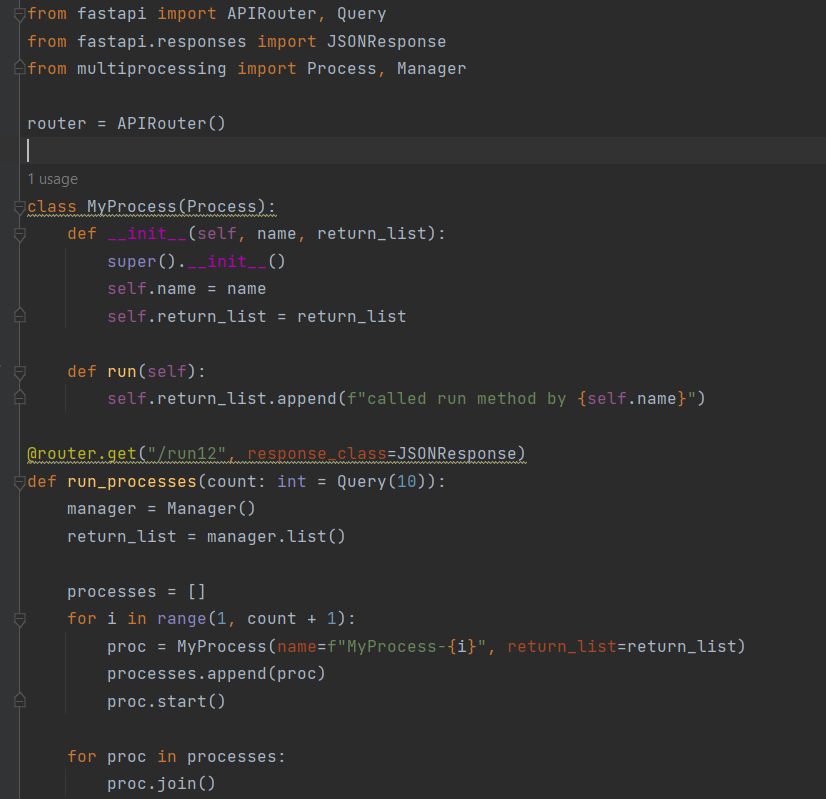
* **long\_running\_task()**

این تابع یک حلقه بی‌نهایت اجرا می‌کند (while True) و هر ثانیه یکبار با استفاده از time.sleep(1) منتظر می‌ماند. این تابع برای شبیه سازی یک فرآیند کارآمد طولانی‌مدت استفاده می‌شود که نیاز به مدیریت وقفه و اجرای طولانی‌مدت دارد.

* **run\_process(duration: int = Query(10))**
  + این تابع به عنوان یک endpoint در وب سرویس تعریف شده است که درخواست‌ها را مدیریت می‌کند. پارامتر duration به عنوان ورودی دریافت می‌شود که به عنوان زمان حداکثر برای اجرای فرآیند بدون پس‌زمینه در نظر گرفته می‌شود.
* یک فرآیند جدید با استفاده از کلاس Process از multiprocessing ایجاد می‌شود. تابع long\_running\_task به عنوان وظیفه‌ای که در فرآیند جدید اجرا خواهد شد، به عنوان target انتخاب شده است.
* لاگ‌های اجرای فرآیند (مانند وضعیت فعلی، وضعیت اجرا، وضعیت ترمینیت، وضعیت جوین و کد خروجی) در لیست logs ذخیره می‌شود.
* فرآیند با فراخوانی proc.terminate() ترمینه شده و در نهایت با استفاده از proc.join()، کد خروجی فرآیند در proc.exitcode ذخیره می‌شود.
* نتیجه نهایی به صورت JSON شامل تمام لاگ‌های جمع‌آوری شده برگردانده می‌شود.

این روش به شما این امکان را می‌دهد که عملیات‌های طولانی‌مدت را به صورت پس‌زمینه و بدون تحت فشار کاربران از طریق واسط API انجام دهید.

**Defining processes in a subclass**





* **MyProcess**:

این کلاس یک زیرکلاس از Process است که برای اجرای یک فرآیند خاص تعریف شده است.

در تابع \_\_init\_\_، نام (name) و یک لیست (return\_list) به عنوان ورودی‌ها گرفته می‌شود و به وسیله‌ی تابع سازنده از کلاس پدر Process فراخوانی می‌شود.

در تابع run، که متدی است که به عنوان ورودی استفاده می‌شود، پیغامی که نشان دهنده این است که تابع اجرایی از آن استفاده میکند

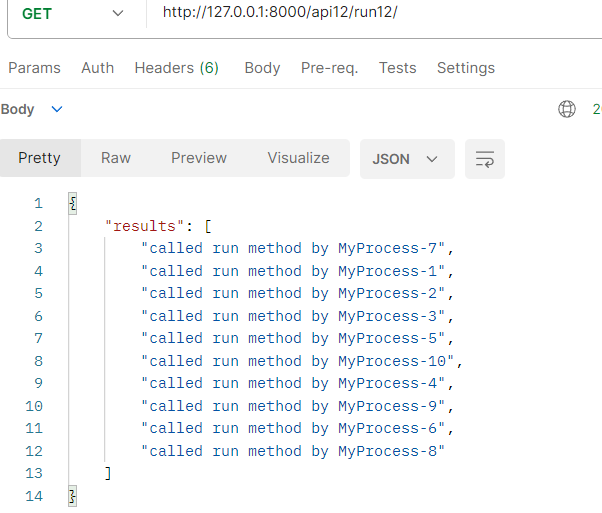
**run\_processes(count: int = Query(10))**:

این تابع به عنوان یک endpoint در وب سرویس تعریف شده است که درخواست‌ها را مدیریت می‌کند. پارامتر count به عنوان تعداد فرآیندها که به عنوان ورودی دریافت می‌شود.

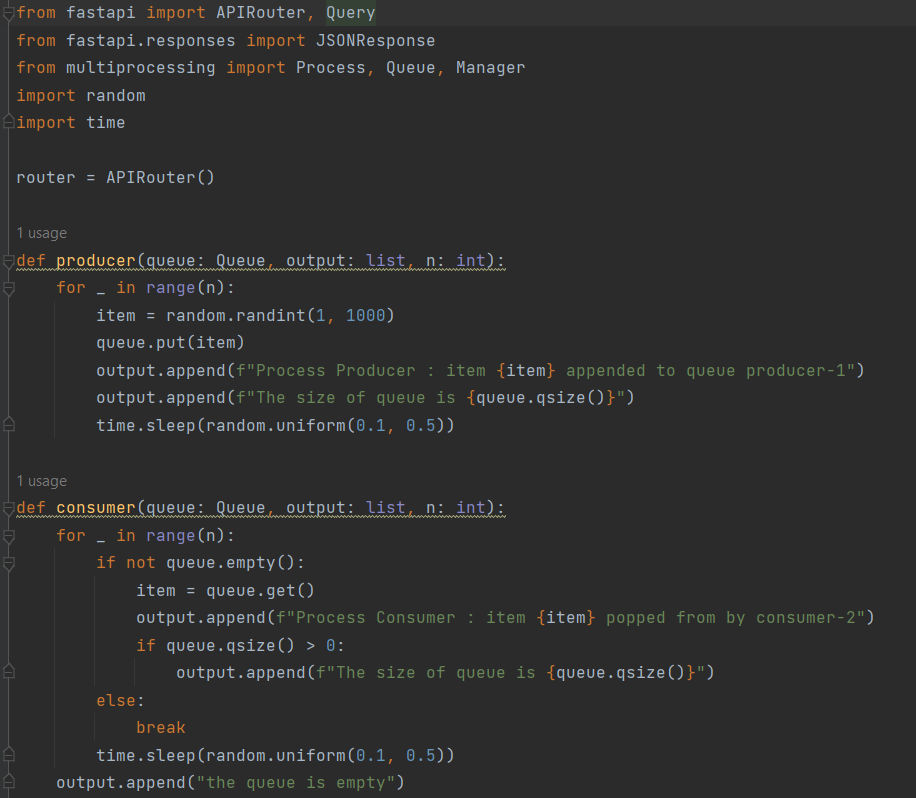
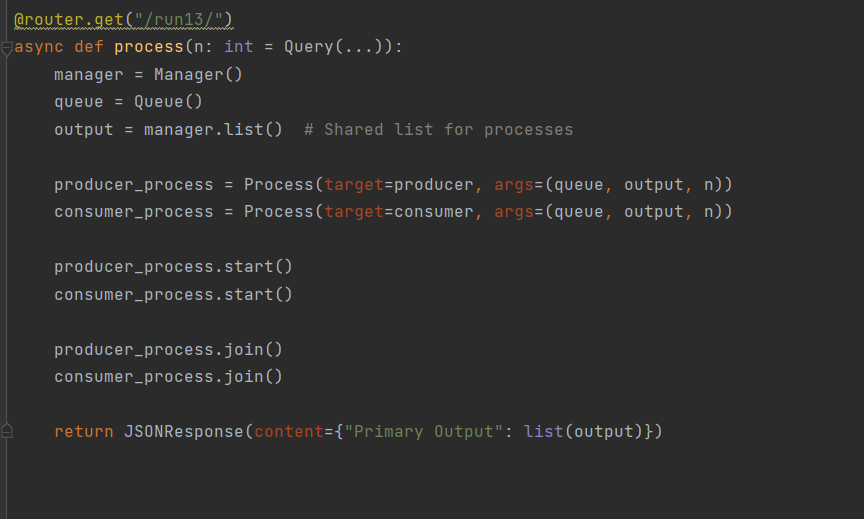
یک Manager ایجاد می‌شود که مسئول مدیریت اشیاء مشترک بین فرآیندها مانند لیست‌ها می‌باشد.

یک Manager با استفاده از Manager ایجاد می‌شود که اشیاء مشترک بین فرآیندها مانند اشیاء استفاده می‌کند. این تابع شامل یک حلقه است که برای هر شماره از به از مقدار و استفاده از فرآینده‌ها استفاده می‌کند. با فرآیندها اجر شدند، داده‌ها بصورت اضافه به است و مشترک است. استفاده می‌شود که بین لیست‌های استفاده است.

این رویکرد به شما این امکان را می‌دهد که چندین فرآیند را به طور موازی اجرا کرده و از لیست‌های مشترک برای جمع‌آوری نتایج و اطلاعات بین آن‌ها استفاده کنید.



**Using a queue to exchange data**

* **producer(queue: Queue, output: list, n: int)**:

این تابع برای تولید داده‌ها به صورت تصادفی و قرار دادن آن‌ها در صف (queue) استفاده می‌شود.

هر داده‌ای که تولید شود، به queue اضافه می‌شود و یک پیام در output نمایان می‌شود که نمایش دهد که کدام عضو را در صف اضافه شده است.

همچنین اندازه‌ی فعلی صف نیز نمایش داده می‌شود.

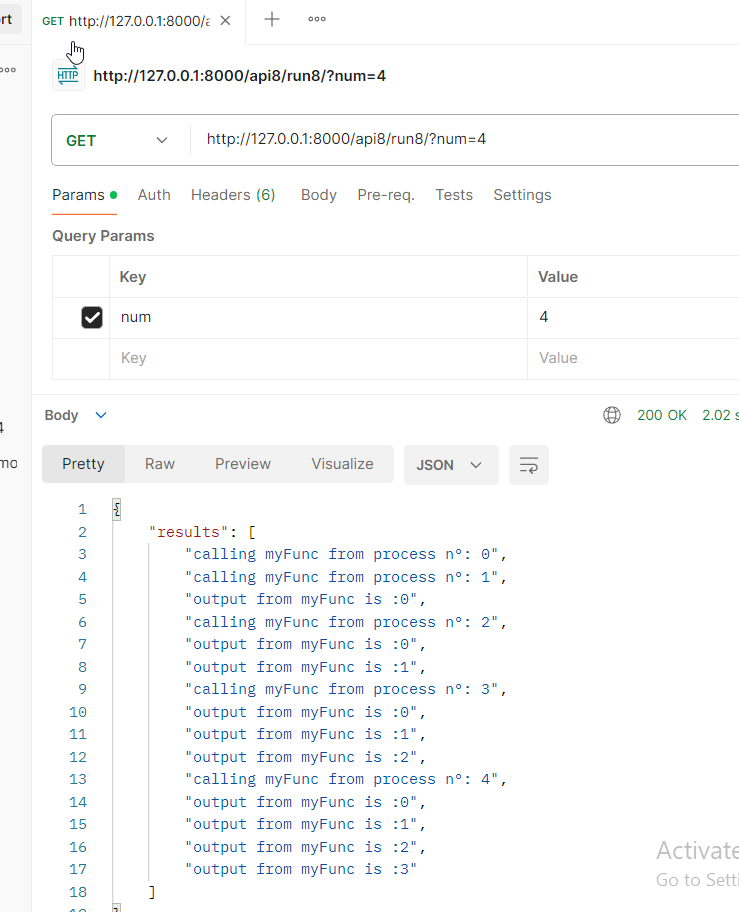
بین هر عملیات تولید داده، یک تاخیر تصادفی برای شبیه‌سازی عملیات ایجاد شده است.

* **consumer(queue: Queue, output: list, n: int)**:

در غیر این صورت، حلقه مصرف‌کننده متوقف می‌شود (break) و پیامی در output نمایش داده می‌شود که نشان دهنده‌ی خالی شدن صف است.

بین هر عملیات مصرف داده، یک تاخیر تصادفی برای شبیه‌سازی عملیات انجام می‌شود (time.sleep(random.uniform(0.1, 0.5))).

این تابع در کل از queue برای مصرف داده‌ها استفاده می‌کند و با استفاده از توابع queue.put() و queue.get() داده‌ها را به صورت تولید کننده و مصرف کننده از صف حذف می‌کند.



**Synchronizing processes**

**توابع test\_with\_barrier و test\_without\_barrier**

* **test\_with\_barrier(barrier, queue, label)**:

این تابع یک فرآیند مولتی پراسسینگ است که با استفاده از barrier منتظر می‌ماند تا همه‌ی فرآیندها به نقطه‌ی بارییر برسند و سپس ادامه می‌دهد.

وقتی که به نقطه‌ی بارییر می‌رسد، زمان کنونی را ثبت کرده و پیامی را به queue اضافه می‌کند که نشان دهنده‌ی زمان و وضعیت فعلی است.

* **test\_without\_barrier(queue, label)**:

این تابع نیازی به بارییر ندارد و بلافاصله زمان کنونی را ثبت کرده و پیامی را به queue اضافه می‌کند که نشان دهنده‌ی زمان و وضعیت فعلی است.

**آرایه‌ی فرآیندها (Process objects)**چهار فرآیند ایجاد می‌شود:

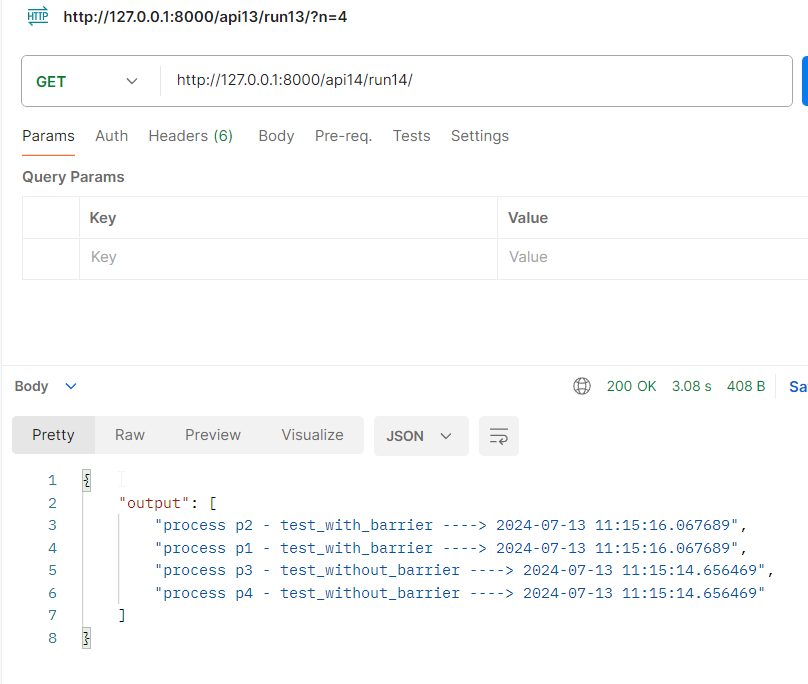
درواقع و p3,p4 دو فرآیند که تابع test\_without\_barrier را فراخوانی می‌کنند بدون استفاده از barrier.

وp1 و p2: دو فرآیند که تابع test\_with\_barrier را فراخوانی می‌کنند با استفاده از barrier.

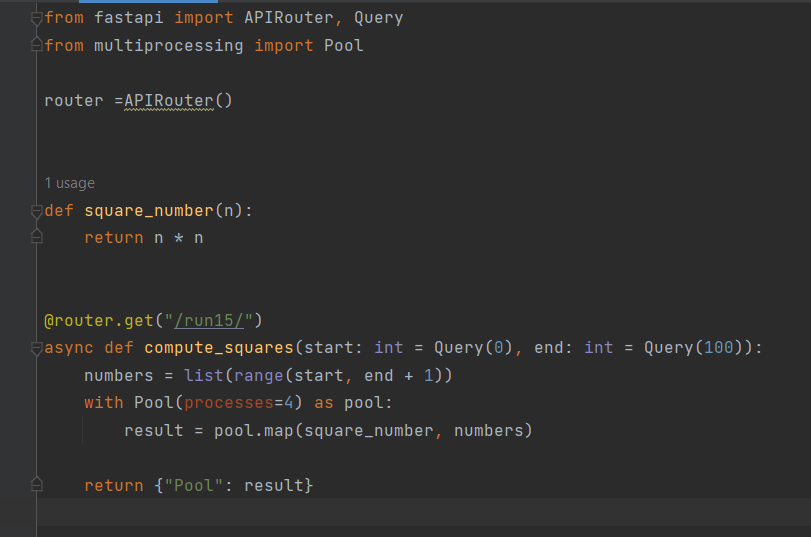
هر فرآیند با فراخوانی start() شروع می‌شود و با فراخوانی join() منتظر اتمام آن می‌ماند.

پس از اتمام همه‌ی فرآیندها، خروجی‌ها از queue جمع‌آوری می‌شوند و بر اساس نام فرآیندها مرتب می‌شوند.

نهایتاً خروجی به صورت مرتب شده برگردانده می‌شود به عنوان JSON Response از طریق FastAPI.



**Using a process pool**



* **square\_number(n)**:

یک تابع ساده است که عدد n را به توان دوم می‌رساند و نتیجه را برمی‌گرداند.

* **router**:

یک شیء از APIRouter است که برای تعریف مسیرهای API در FastAPI استفاده می‌شود.

* **@router.get("/run15/")**:

این مسیر برای GET request به /run15/ در API تعریف شده است.

**تابع compute\_squares**

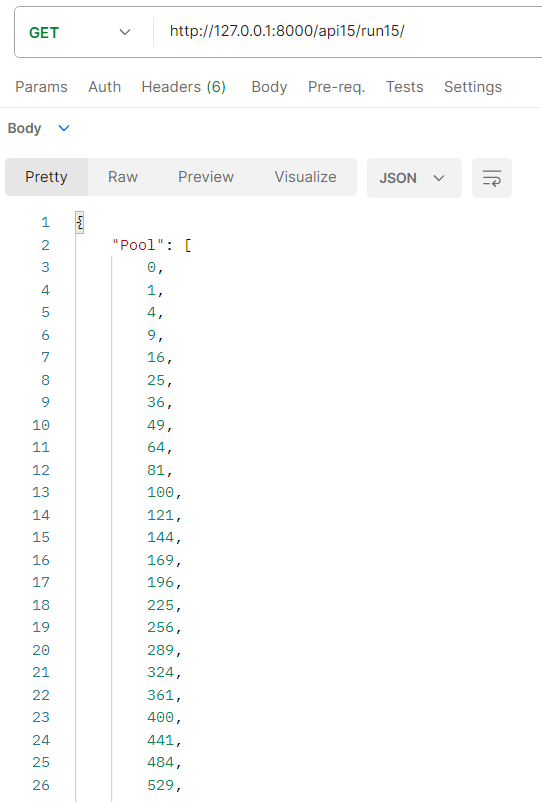
* **compute\_squares(start: int = Query(0), end: int = Query(100))**:

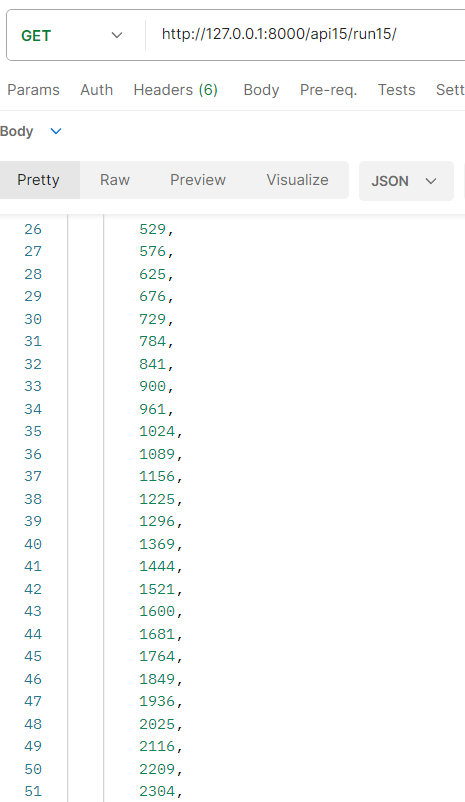
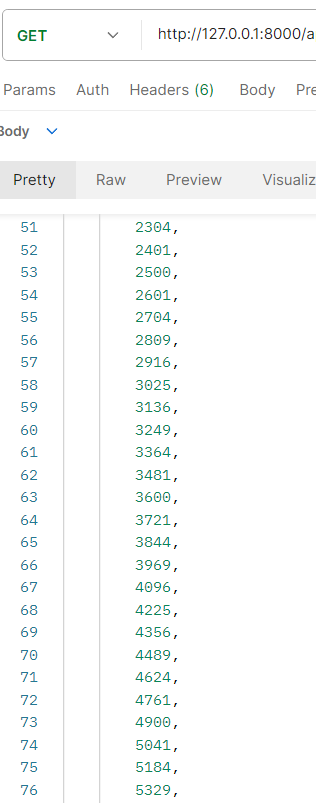
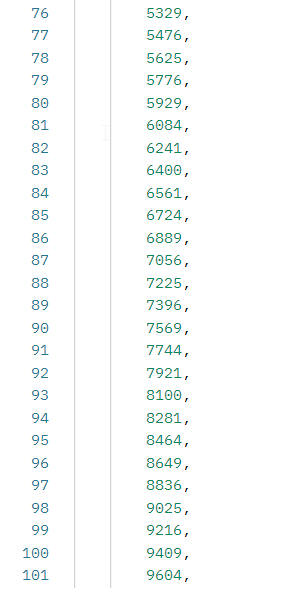
این تابع با استفاده از FastAPI اجرا می‌شود و دو پارامتر start و end را از کوئری دریافت می‌کند.

. Numbers از start تا end را تولید می‌کند و به صورت یک لیست از اعداد تبدیل می‌کند.

یک Pool با چهار فرآیند (processes=4) ایجاد می‌شود.. pool.map(square\_number, numbers)استفاده می‌شود تا تابع square\_number را روی هر عضو از numbers اعمال کرده و نتایج را جمع‌آوری کند

نتایج به صورت یک دیکشنری JSON شامل Pool و لیست نتایج محاسبه شده برگردانده می‌شود.

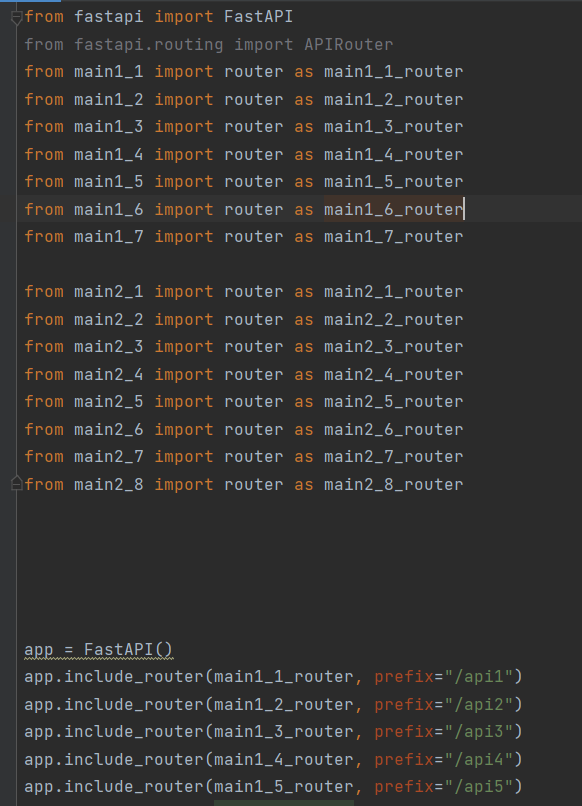


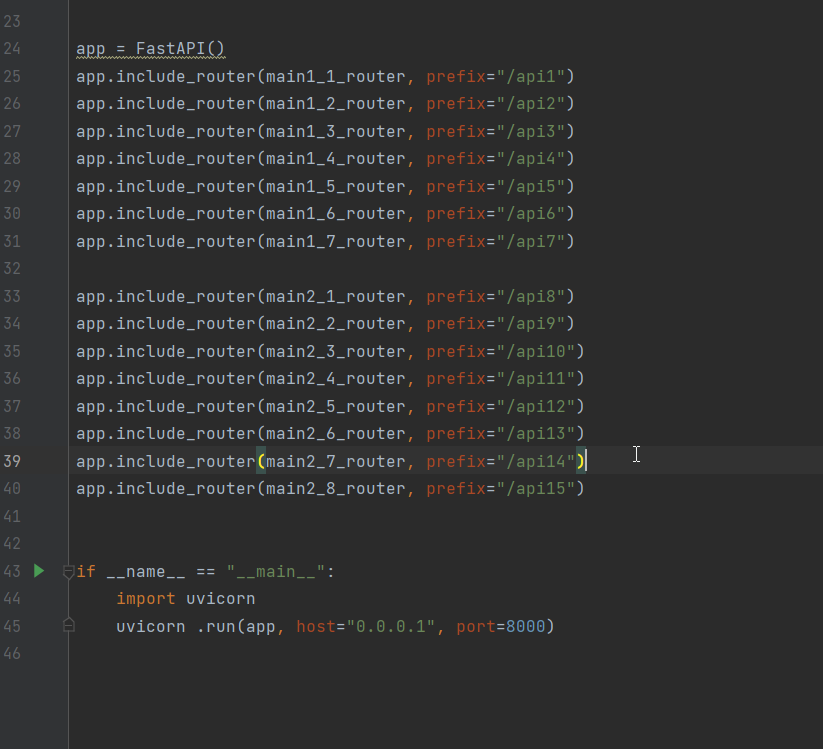
  

............................................................................................................................................................

در بالا به بررسی کدها و سناریو های تمرین پرداختیم توضیحات تکمیلی :

در هر کد با کمک fastapiنوشته شد با استفاده از ماژول روتر بصورت بسط کردن بکار گرفته میشود که در mainمشخص است بصورتی که پس از نوشتن هر کد در main شئ از روتر را می سازد و با استفاده از آن به دنبال اجرای آن با APIRouter است



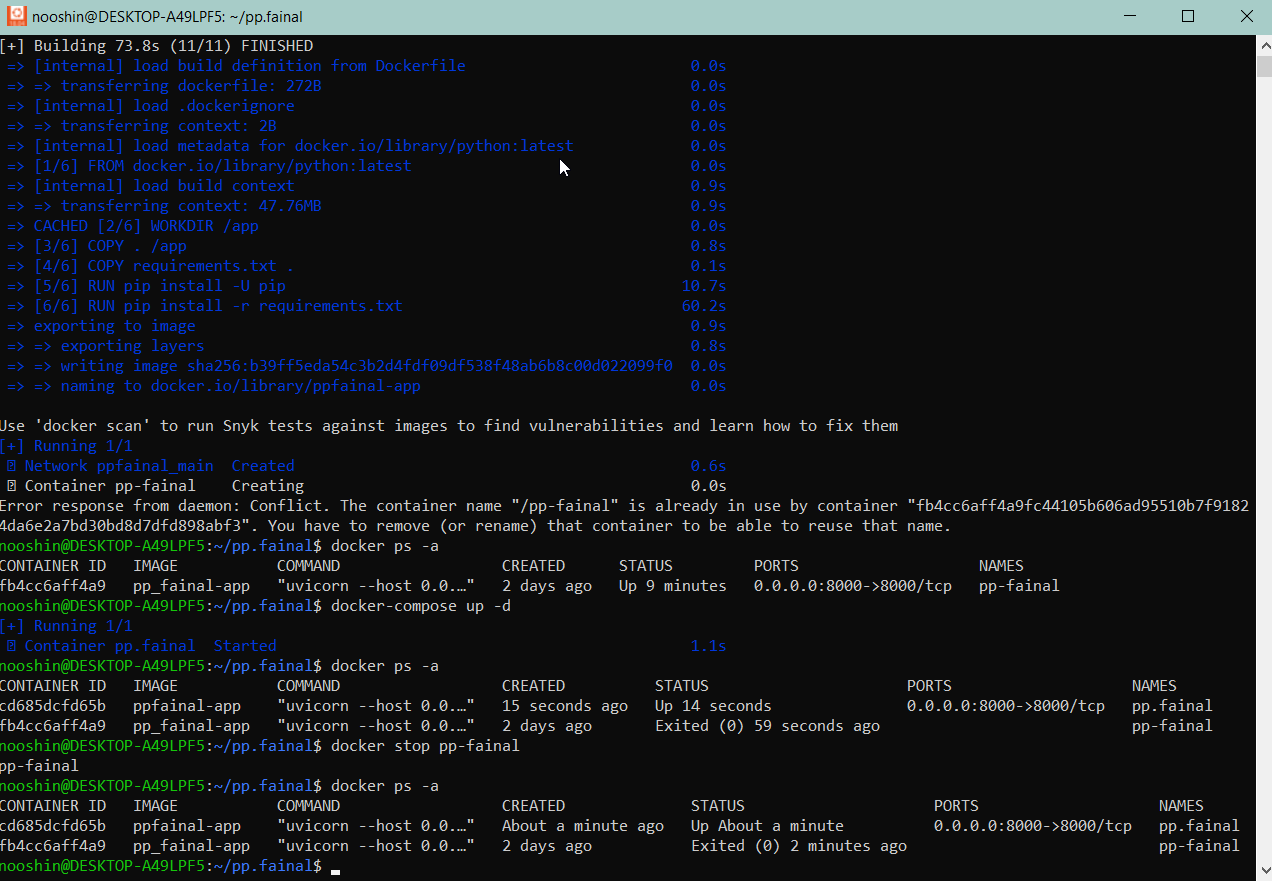


...............................................................................................................................................................................................................................................روند اجرای پروژه :

1.نوشتن هر سناریو

2.داکرایز کردن و ساختن imageاز هر کد با نوشتن Dockerfile,docker-compose.ymlکه محتوای هر کدام در فایل هست

سپس با کد :



به ایجاد کانتینرها پرداختیم

و در اخر:

با ابزار Gitbash به pushکردن پروژه در گیت هاب پرداختیم:

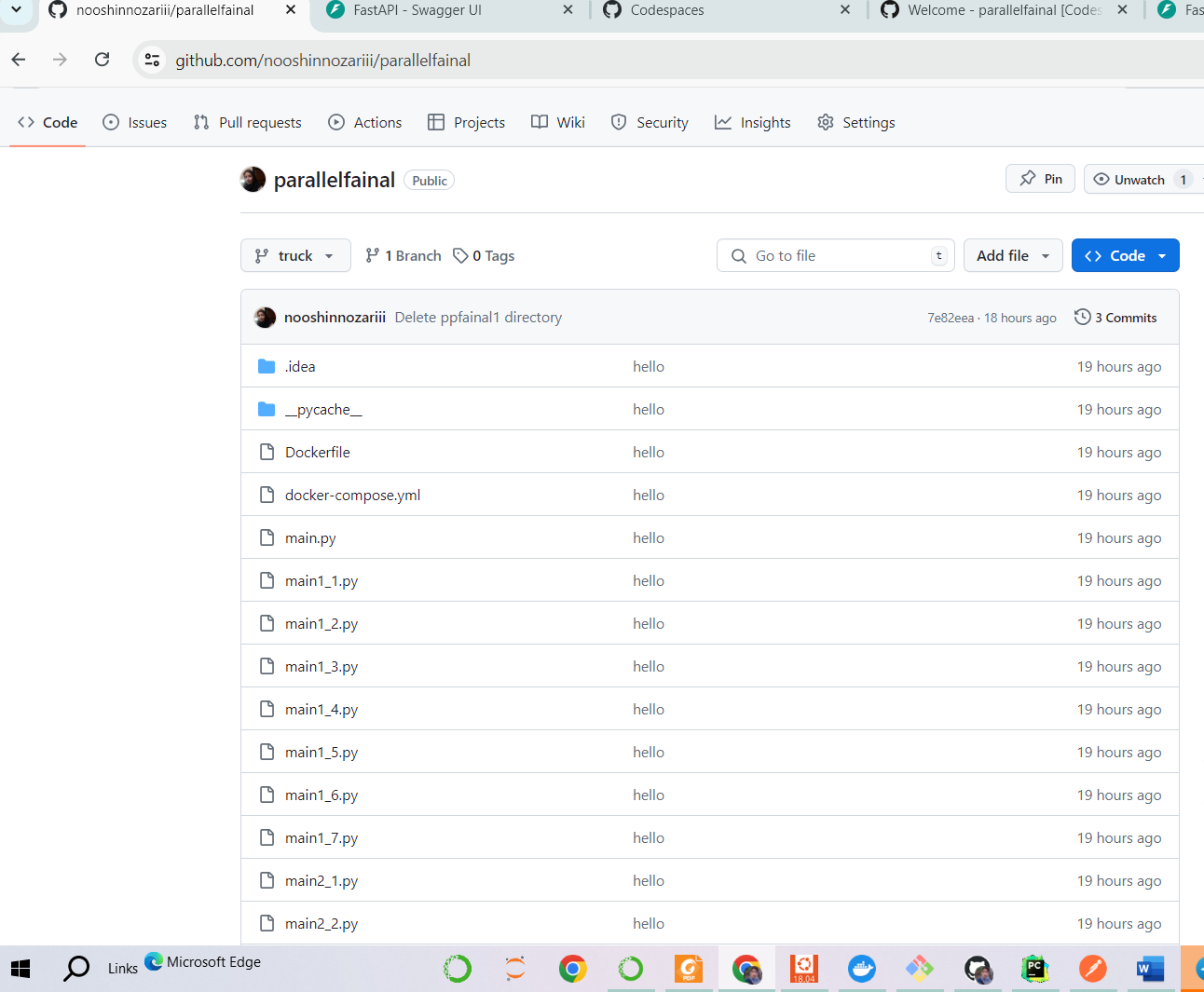
1.git init

2.git add .

3.git commit -m “hello”

4.git remote add origin <http://github.com/nooshinnozariii/paralellfainal>

5.git push -u origin truck



در نهایت برای نمایش عمومی بر روی سرور:

وارد codespaceدر گیت هاب میشویم سپس وارد اکانت گیت هاب خود در codespaceمیشویم و با مراحل:

