**به نام خدا**

**مبانی رایانش توزیع شده**

**گزارش پروژه دوم**

**مرضیه حریری 810199404**

**فاطمه زهرا برومندنیا 810100094**

**نیما تاجیک 810100104**

**امیرحسین راحتی 810100144**

**مقدمه**

**در این تمرین ، هدف ایجاد یک سیستم رزرو بلیت با سرعت بالا به کمک زبان Golang است . این زبان به خاطر ویژگی های خاص خود ، توانایی multi-threading را فراهم میکند . این سیستم باید شرایط خاص مثل race condition ، fairness ، performance و ... را کنترل کند.**

**در ادامه به بررسی کد میپردازیم**

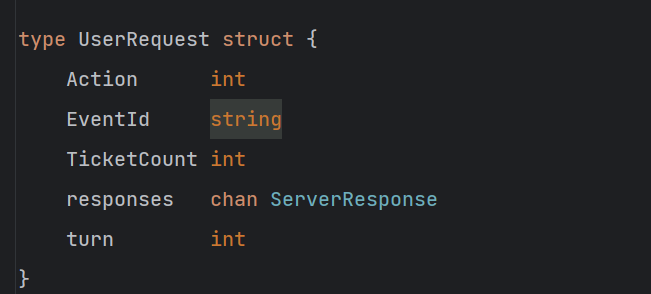
**Structures**

**در این تمرین چندین استراکت برای ذخیره انواع دیتا ها مورد استفاده قرار گرفته است که هریک به اختصار توضیح داده خواهد شد.**

* **UserRequest**

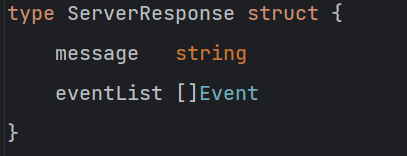
**در این ساختار ،دو حالت داریم . یکی برای رزرو و دیگری برای مشاهده لیست که توسط عضو اول ساختار مشخص میشود**

**آیدی رویداد مورد نیاز کاربر ، تعداد بلیط ها ، پاسخ درخواست و نوبت این درخواست در بین درخواست های رقابتی(که ممکن است دچار race condition شوند) مشخص میشود.**



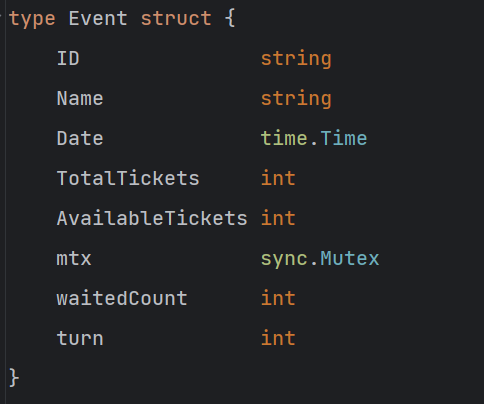
* **ServerResponse**

**در این ساختار ساده ، یک پیام و لیستی از رویداد ها به عنوان پاسخ سرور برگردانده میشود که در هر یک از حالات درخواست ها مقادیر فیلد های آن متفاوت خواهد بود.**



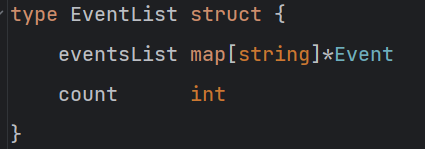
* **Event**

**این ساختار ، مقادیر و متغیر های یک رویداد را نگه داری میکند .دو فیلد آخر برای تعیین نوبت رزرو بین thread های هم روند است .**



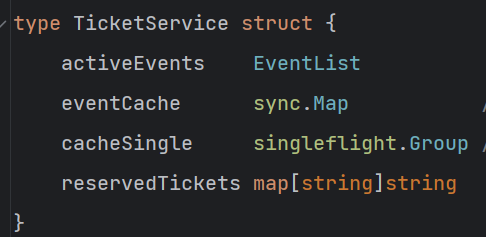
* **EventList**

**لیستی از رویداد ها به همراه تعداد آن ها را نگه میدارد**



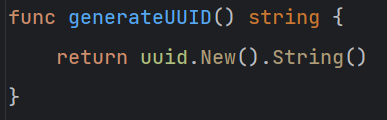
* **TicketService**

**این ساختار ، لیستی از رویداد ها ، به همرا فیلد هایی برای caching و مپی از بلیط های رزرو شده ذخیره میکند.**



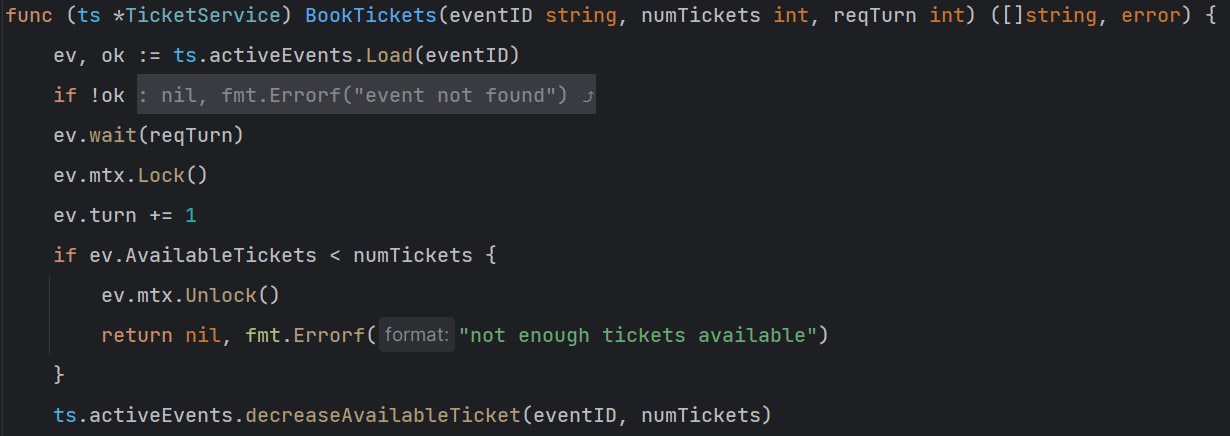
**بررسی ساختار برنامه و توابع مهم**

**در این تمرین برای ایجاد آیدی های یکتا برای بلیط ها ، از کتابخانه uuid استفاده کردیم که تضمین میدهد آیدی های یکتا بدهد**

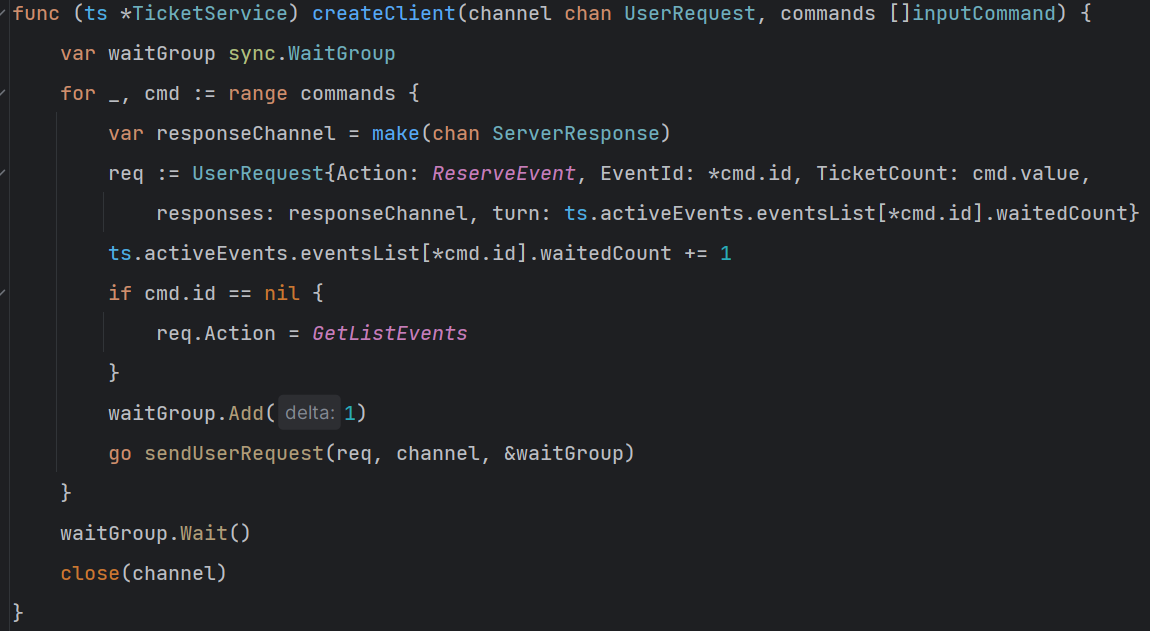


**توابع store() و load() وظیفه ذخیره یا لود کردن یک رویداد از لیست رویداد هارا به عهده دارند.**

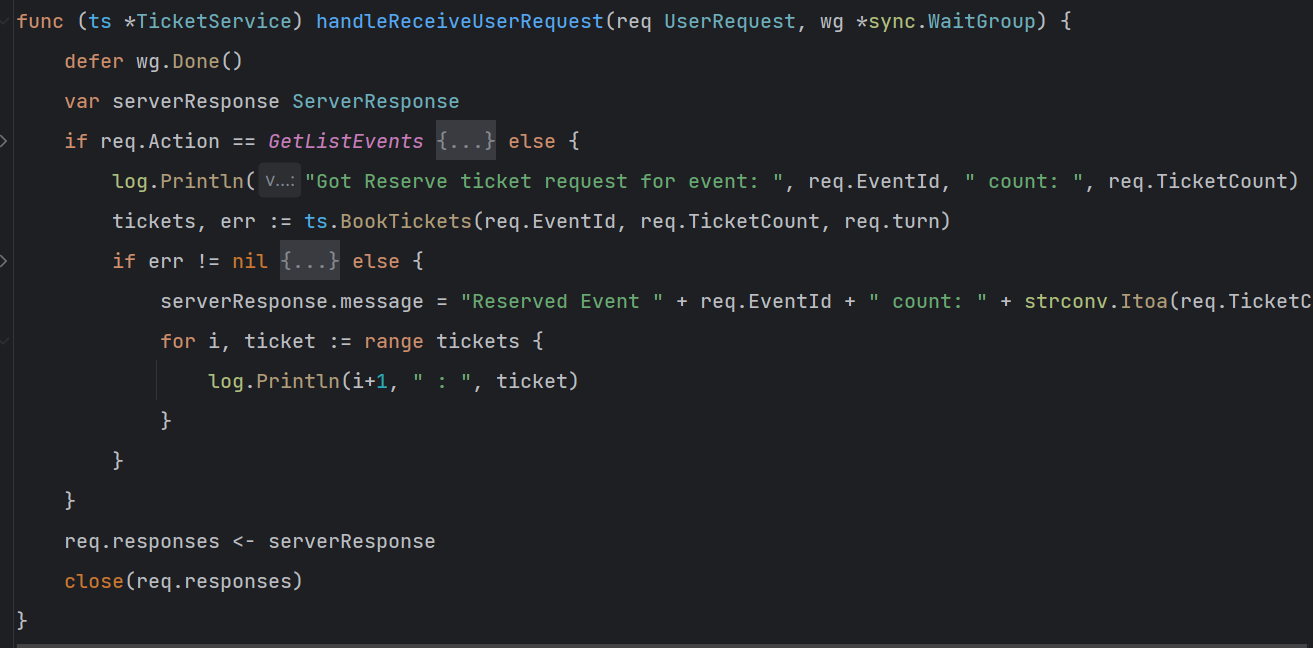
**تابع bookTicket وظیفه رزرو تیکت به تعداد دلخواه بر روی یک رویداد داده شده را دارد. نقاط بحرانی این تابع بوسیله mutex مربوط به آن رویداد محافظت میشود که در ادامه بیشتر بررسی خواهد شد.**



**تابع CreateClient مسئولیت دریافت درخواست هایی که از interface (ترمینال یا فایل ) خوانده میشود را دارد و آن ها را روی یک چنل بویسله thread های مختلف ارسال میکند.**



**تابع handleRecieveUserRequest وظیفه بررسی درخواست ها را دارد و باید هرکدام را اجرا کند و نتیجه را در channel مربوط به پاسخ برای کاربر ارسال کند .**



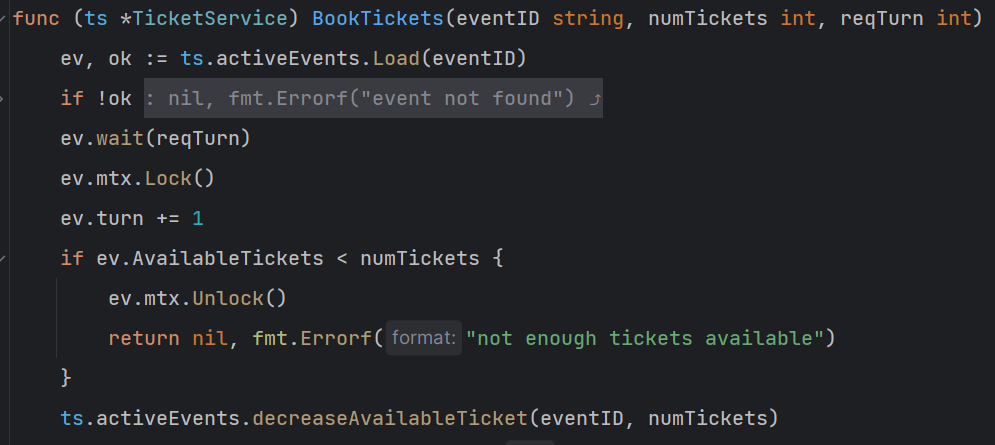
**روند اجرای برنامه**

**ابتدا همه درخواست ها توسط تابع interface خوانده میشود و در قالب یک لیست برگردانده میشود . سپس این درخواست ها در یک تابع createClinet به عنوان درخواست client ها بر روی چندین thread روی یک channel اشتراکی ارسال میشود . سپس این درخواست ها به وسیله چندین consumer thread دریافت و شروع به اجرا شدن می کنند و پاسخ را به روی channel اختصاصی پاسخ که توسط client در درخواست ارسال شده بود قرار میدهد.**

**بررسی fairness و RaceCondition در برنامه**

**برای جلوگیری از race و همچنین fairness نیاز داریم مکانیزمی داشته باشیم که علاوه بر جلوگیری از دسترسی به قسمت های بحرانی توسط دیگر thread ها ، آن ها را به گونه ای مدیریت کند که هیچ کدام خارج از نوبت درخواستی ندهند و درخواست ها به ترتیب آمدنشان مدیریت شوند .**

**برای این منظور ، ابتدا بخش های critical کد را شناسایی میکنیم . قسمت زیر یکی از بخش های بحرانی برنامه است :**



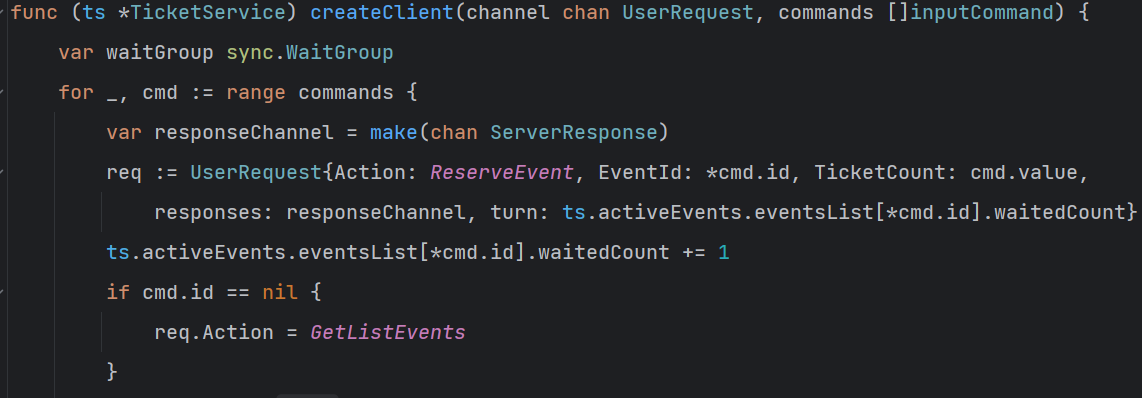
**در این قسمت ، ابتدا چک میشود که تیکت های مورد نیاز موجود باشد ، سپس آن تعداد مورد نیاز را در صورت وجود از تعداد کل کم میکند .**

**واضح است که این بخش اگر همزمان توسط چند رشته مورد استفاده و نوشتن قرار بگیرد ،**

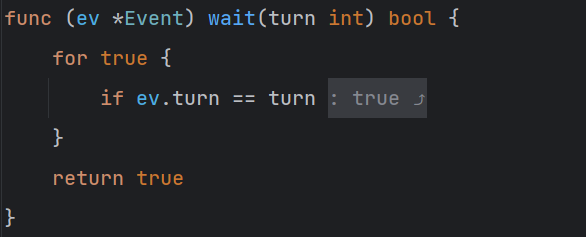
**Data consistency خواهیم داشت و رزرو به درستی انجام نمیشود. برای همین از این بخش توسط mutex متعلق به event محافظت میکنیم تا از دسترسی همزمان به یک رویداد جلوگیری کنیم .**

**حال نیاز داریم که شرایطی را بوجود بیاوریم که هر درخواست به ترتیب آمدنش بررسی و پردازش شود و خارج از نوبت این کار انجام نشود**

**برای این کار ، یک متغیر اضافی turn تعریف میکنیم که مشخص میکند در هر لحظه نوبت کدام درخواست بر روی آن event است که اجرا شود . یک متغیر waitedCount هم داریم که این متغیر، در زمان ایجاد ، مقدار صفر دارد . با بوجود آمدن هر درخواست روی یک event مقدار آن یکی زیاد میشود . از مقدار این متغیر برای نوبت دهی به request ها استفاده میکنیم . عملیات آپدیت و استفاده از این متغیر در تابع createClient قابل مشاهده است :**



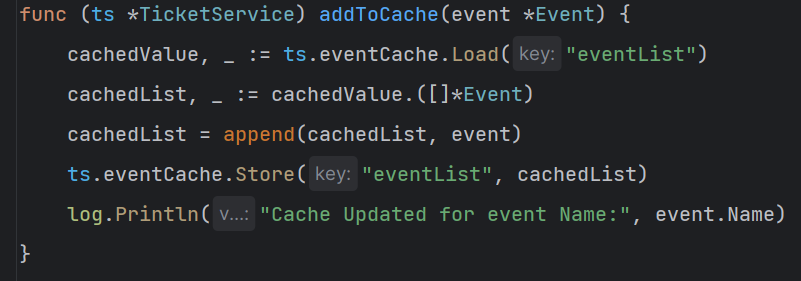
**به کمک تابع wait() ، هر درخواست را ملزم میکنیم تا رسیدن به نوبتش صبر کند .**



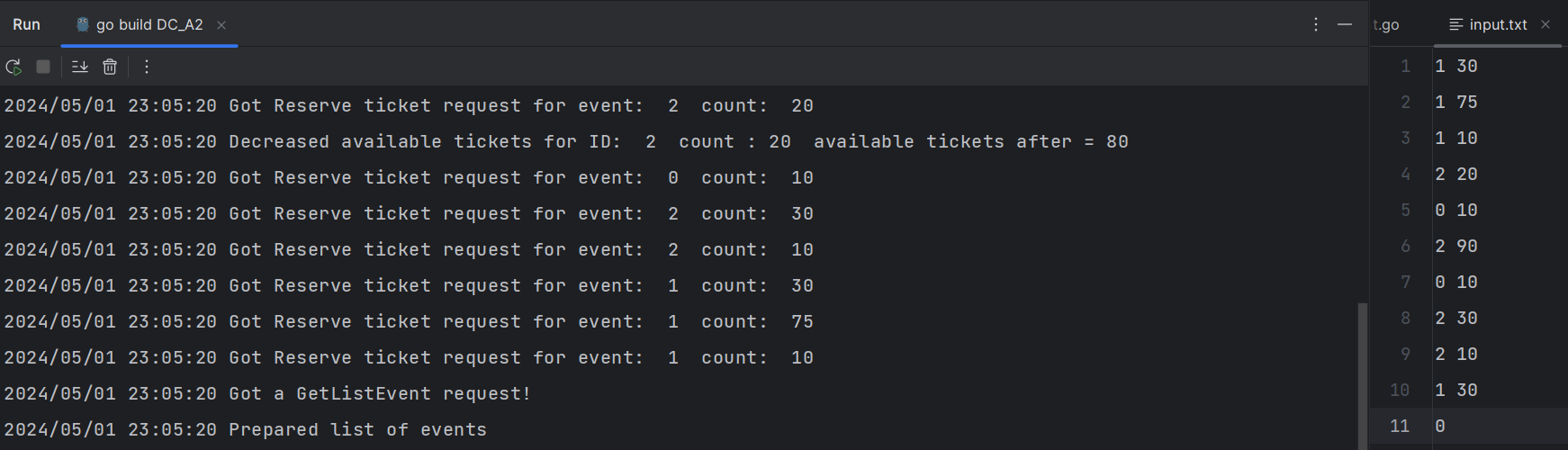
**هر درخواست بعد از تمام شدن کارش با قسمت بحرانی ، مقدار turn را یکی زیاد میکند که باعث میشود درخواست بعدی بتواند وارد شود و کار خود را انجام دهد.**

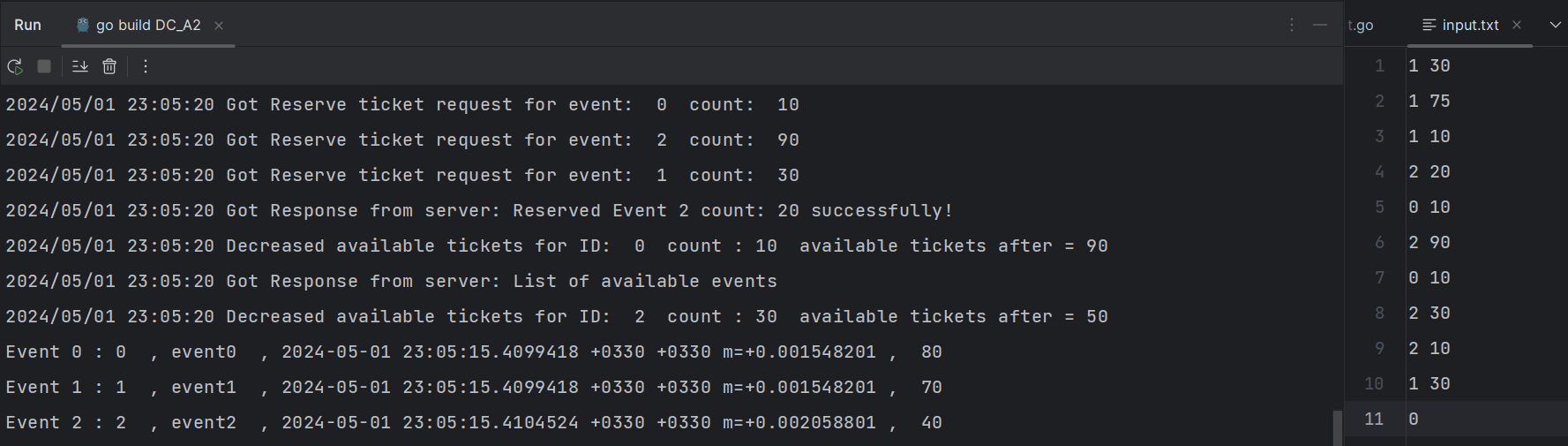
**پیاده سازی مکانیزم Caching**

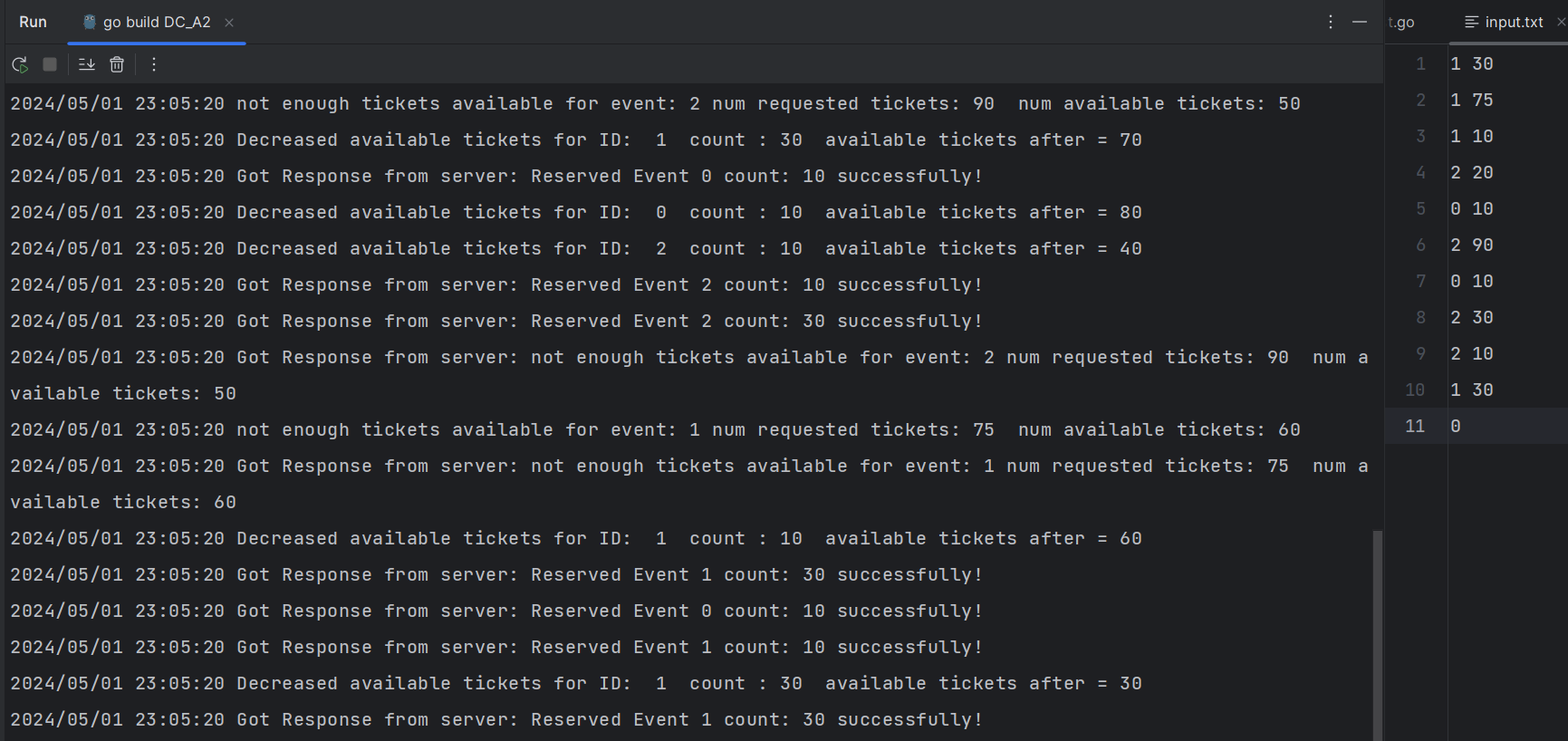
**برای پیاده سازی این مکانیزم داخل استراکت TicketService دو متغیر اضافه شدند. یکی eventCache که لیست event ها را با کلید “eventList” ذخیر میکند و cacheSingle که از نوشتن همزمان چند go cache نیز آپدیت می شود. در کش ما پوینتری به داده ایونت داریم در نتیجه با آپدیت شدن ایونت کش نیز آپدیت می شود. Routine بر روی آن جلوگیری می کند. همچنین در هر زمان که تغییری بر روی event list ها رخ دهد. تابع زیر این کار را انجام خواهد داد:**



**روند اجرای برنامه**







**در سه تصویر بالا ، به ازای ورودی سمت راست صفحه که تعیین میکند از هر رویداد چه تعداد بلیت رزرو شود ، خروجی درخواست ها در سمت چپ قابل مشاهده است . با استفاده از مکانیزم aging و priority پیاده سازی شده ، هیچ درخواستی که دیرتر ارسال شده ، قبل از درخواست زودتر، انجام نمیشود زیرا برای هرکدام turn درنظر گرفته شده است .**

**درنهایت اطلاعات رویداد ها در شکل زیر قابل مشاهده است . تیکت های باقیمانده به درستی قابل مشاهد هستند.**

