In large enterprises, business workflows often require distributed components for seamless scaling and maintenance. Hence it is essential to develop effective connectivity and communication between such elements. Event-driven architecture plays a vital role in such scenarios.

در شرکت‌های بزرگ، گردش‌های کاری تجاری اغلب به اجزای توزیع‌شده برای مقیاس‌بندی و نگهداری بدون درز نیاز دارند. از این رو ایجاد ارتباط و ارتباط موثر بین این عناصر ضروری است. معماری رویداد محور در چنین سناریوهایی نقش حیاتی دارد.

**What is an event?**

An event can be described as ‘something that has happened or a significant change in the state’. Events are immutable, i.e. they cannot be changed or deleted, and they are ordered in sequential order of their occurrence. A component for other components creates a notification to inform them about the occurrence of an event. It mainly consists of information about the possibilities and context of their occurrence, such as location, time, etc.

یک رویداد را می توان به عنوان "چیزی که اتفاق افتاده است یا تغییر قابل توجهی در وضعیت" توصیف کرد. رویدادها تغییرناپذیرند، یعنی نمی‌توان آن‌ها را تغییر داد یا حذف کرد و به ترتیب ترتیب وقوعشان مرتب می‌شوند. یک مؤلفه برای سایر مؤلفه ها یک اعلان ایجاد می کند تا آنها را در مورد وقوع یک رویداد مطلع کند. عمدتاً شامل اطلاعاتی درباره احتمالات و زمینه وقوع آنها مانند مکان، زمان و غیره است.

**What is event-driven architecture?**

An event-driven architecture uses events to trigger and communicate between decoupled services or components. EDA makes it possible to exchange information between the systems in real-time as events occur instead of periodically polling for updates. It is commonly used in many modern applications built using microservices.

یک معماری رویداد محور از رویدادها برای راه اندازی و برقراری ارتباط بین سرویس ها یا اجزای جدا شده استفاده می کند. EDA امکان تبادل اطلاعات بین سیستم ها را به صورت بلادرنگ به عنوان رویدادها به جای نظرسنجی دوره ای برای به روز رسانی، فراهم می کند. معمولاً در بسیاری از برنامه های مدرن که با استفاده از میکروسرویس ها ساخته شده اند استفاده می شود.

Event-driven architectures have three main components:  
1) Event producers: publish events to the router  
2) Event routers: filters, processes and routes the events to consumers  
3) Event consumers: receive the event and take action on them

معماری های رویداد محور دارای سه جزء اصلی هستند:

1) تولیدکنندگان رویداد: رویدادها را در روتر منتشر کنید

2) روترهای رویداد: فیلتر، پردازش و مسیریابی رویدادها به مصرف کنندگان

3) مصرف کنندگان رویداد: رویداد را دریافت کرده و نسبت به آنها اقدام کنید

Producer and consumer in event-driven architecture are decoupled; hence they can be scaled, updated, and deployed independently.  
  
Event-driven architecture is mainly implemented using the publish/subscribe message pattern.  
In the publish/subscribe model, consumers are called subscribers and producers are called publishers. Subscribers sign up for the topics/channels of their interest. Publishers publish events to the topics, and the subscribers receive the events that interest them. Hence, this pattern is the most suitable framework for streaming real-time data.

تولید کننده و مصرف کننده در معماری رویداد محور از هم جدا شده اند. از این رو آنها را می توان مقیاس، به روز رسانی و به طور مستقل مستقر کرد.

معماری رویداد محور عمدتاً با استفاده از الگوی پیام انتشار/اشتراک اجرا می‌شود.

در مدل انتشار/اشتراک، مصرف کنندگان را مشترک و تولیدکنندگان را ناشر می نامند. مشترکین برای موضوعات/کانال های مورد علاقه خود ثبت نام می کنند. ناشران رویدادها را به موضوعات منتشر می کنند و مشترکین رویدادهای مورد علاقه خود را دریافت می کنند. از این رو، این الگو مناسب ترین چارچوب برای جریان داده های بلادرنگ است.

Let us understand event-driven architecture using a real-world example of a food delivery application. You have ordered your favourite food from a restaurant. You are anxiously waiting for the food. Now with the traditional architecture, i.e., request-driven approach, you wouldn't know the status of your order since to get the status you need to send a request. Thus, you will have to click a button on the application, which will update the current status of your order. Therefore, you are periodically polling to get the updated status. The polling, in this case, is very annoying for you and the system; it is costly and consumes a lot of bandwidth with no positive result.

اجازه دهید معماری رویداد محور را با استفاده از یک مثال واقعی از برنامه تحویل غذا درک کنیم. شما غذای مورد علاقه خود را از یک رستوران سفارش داده اید. شما مشتاقانه منتظر غذا هستید. اکنون با معماری سنتی، یعنی رویکرد مبتنی بر درخواست، از وضعیت سفارش خود اطلاعی ندارید زیرا برای دریافت وضعیت باید درخواست ارسال کنید. بنابراین، شما باید روی یک دکمه روی برنامه کلیک کنید، که وضعیت فعلی سفارش شما را به روز می کند. بنابراین، شما به صورت دوره ای نظرسنجی می کنید تا وضعیت به روز شده را دریافت کنید. نظرسنجی در این مورد برای شما و سیستم بسیار آزاردهنده است. پرهزینه است و پهنای باند زیادی را بدون نتیجه مثبت مصرف می کند.

Now let us view this situation with an event-driven architecture approach.   
When you order food using your application, a food order event is published. The restaurant which has subscribed to this event receives it and accepts your order. It publishes a status event that your application receives, which then updates the status of your order. So now you can view the status that the restaurant has accepted your order and food is being prepared. When your food is prepared, the restaurant publishes another food-ready event, and the delivery person subscribing to this event picks up the order and starts his ride. Simultaneously order status on your application is updated. An event is triggered with each update in the rider’s location, and you are updated in real-time. Thus with this approach, you know the current status of your order without continuously polling for the same.

حال اجازه دهید این وضعیت را با رویکرد معماری رویداد محور ببینیم.

هنگامی که با استفاده از برنامه خود غذا سفارش می دهید، یک رویداد سفارش غذا منتشر می شود. رستورانی که در این رویداد مشترک شده است آن را دریافت کرده و سفارش شما را می پذیرد. یک رویداد وضعیتی را منتشر می کند که برنامه شما دریافت می کند و سپس وضعیت سفارش شما را به روز می کند. بنابراین اکنون می توانید وضعیتی را مشاهده کنید که رستوران سفارش شما را پذیرفته و غذا در حال آماده شدن است. وقتی غذای شما آماده شد، رستوران یک رویداد آماده غذایی دیگر منتشر می‌کند و تحویل‌دهنده مشترک این رویداد سفارش را تحویل می‌گیرد و سواری خود را شروع می‌کند. به طور همزمان وضعیت سفارش در برنامه شما به روز می شود. با هر به‌روزرسانی در مکان سوار، رویدادی فعال می‌شود و شما در زمان واقعی به‌روزرسانی می‌شوید. بنابراین با این رویکرد، شما از وضعیت فعلی سفارش خود بدون نظرسنجی مداوم برای آن مطلع هستید.

**Benefits of EDA:**

Loose coupling: Systems using EDA have components that are loosely coupled. This decoupling helps in the logical separation of production and consumption of events. Producers are not concerned with how the events are consumed, whereas consumers are not worried about producing the events. Because of this loose coupling, different components or microservices can be developed in other languages or use different technologies. It also helps in application scaling. Producers or consumers can be added or removed as per the requirements without any logical change in other components.

اتصال شل: سیستم هایی که از EDA استفاده می کنند دارای اجزایی هستند که به طور سست کوپل شده اند. این جداسازی به تفکیک منطقی تولید و مصرف رویدادها کمک می کند. تولیدکنندگان نگران نحوه مصرف رویدادها نیستند، در حالی که مصرف کنندگان نگران تولید رویدادها نیستند. به دلیل این اتصال شل، اجزا یا میکروسرویس های مختلف را می توان به زبان های دیگر توسعه داد یا از فناوری های مختلف استفاده کرد. همچنین به مقیاس بندی برنامه کمک می کند. تولیدکنندگان یا مصرف کنندگان را می توان بر اساس الزامات بدون تغییر منطقی در سایر اجزاء اضافه یا حذف کرد.

Fault tolerance: Eventing is asynchronous, which infers that events are published as and when they happen. Event consuming service or component subscribes to these events. So when any consumer goes down, the application will continue running in its absence. Events are queued in the broker so that when the consumer recovers from the failure, it can pick up the pending events.  
Better real-time user experiences: Event-driven APIs lead to a better interactive experience for the end-user with modern interactivity demands by abstracting away many of the responsibilities previously assigned to users. Of course, this may increase complexity at the producer, but the output is worth it.

تحمل خطا: رویداد ناهمزمان است، که استنباط می‌کند که رویدادها به محض وقوع و زمان انتشار منتشر می‌شوند. سرویس یا مؤلفه مصرف کننده رویداد در این رویدادها مشترک می شود. بنابراین وقتی هر مصرف کننده ای از کار بیفتد، برنامه در غیاب آن به کار خود ادامه می دهد. رویدادها در کارگزار در صف قرار می گیرند تا زمانی که مصرف کننده از شکست بهبود یافت، بتواند رویدادهای معلق را دریافت کند.

تجربه‌های کاربر در زمان واقعی بهتر: APIهای رویداد محور با حذف بسیاری از مسئولیت‌هایی که قبلاً به کاربران محول شده بود، منجر به تجربه تعاملی بهتری برای کاربر نهایی با نیازهای تعاملی مدرن می‌شوند. البته، این ممکن است پیچیدگی را در تولید کننده افزایش دهد، اما خروجی ارزش آن را دارد.

Cost-effective: Event-driven architecture is pushed based on the producer pressing the message in the queue and the consumer receiving it. This way the continuous polling to check an event is not required. The CPU consumed is less, and the network bandwidth usage is reduced.

مقرون به صرفه: معماری رویداد محور بر اساس فشار دادن پیام در صف توسط تولید کننده و دریافت آن توسط مصرف کننده انجام می شود. به این ترتیب نظرسنجی مداوم برای بررسی یک رویداد مورد نیاز نیست. CPU مصرف شده کمتر است و استفاده از پهنای باند شبکه کاهش می یابد.

مزایای معماری بدون سرور:

زمان آن رسیده است که بفهمیم این خدمات آماده چگونه می توانند به کاهش سریع ایده های زمان عرضه به بازار کمک کنند و ما را از دردسر مدیریت چیزهایی غیر از محصولمان نجات دهند.

Setting up and managing the environment: This step is not needed here. Usually, service providers provide consoles or SDKs to upload a piece of code. Then, this code is provisioned and deployed onto their servers. Also, other BaaS services like database, BLOB storage can be seamlessly accessed via some authentication mechanism.

راه اندازی و مدیریت محیط: این مرحله در اینجا مورد نیاز نیست. معمولا، ارائه دهندگان خدمات کنسول ها یا SDK ها را برای آپلود یک قطعه کد ارائه می کنند. سپس، این کد تهیه شده و بر روی سرورهای آنها مستقر می شود. همچنین، سایر خدمات BaaS مانند پایگاه داده، ذخیره سازی BLOB را می توان به طور یکپارچه از طریق برخی مکانیسم های احراز هویت در دسترس قرار داد.

Security: Unlike traditional approaches, the network configurations and security layer are inherently managed by service providers. These network configurations are well tested and not exposed to the end-user. Any security flaws are easily patched without us even noticing.

امنیت: برخلاف رویکردهای سنتی، پیکربندی‌های شبکه و لایه امنیتی ذاتاً توسط ارائه‌دهندگان خدمات مدیریت می‌شوند. این پیکربندی های شبکه به خوبی آزمایش شده اند و در معرض کاربر نهایی قرار نمی گیرند. هر گونه نقص امنیتی به راحتی بدون اینکه ما متوجه شویم اصلاح می شود.

High level of granularity: With serverless architecture, applications are broken down into smaller pieces. This makes it easier to fix bugs and faster code deployments without downtimes.

سطح بالای دانه بندی: با معماری بدون سرور، برنامه ها به قطعات کوچکتر تقسیم می شوند. این امر رفع اشکالات و استقرار سریع کدها را بدون خرابی آسان تر می کند.

با استفاده از سرور بدون سرور، سازمان درگیر مدیریت سرورها یا پایگاه های داده نیست. از این رو سازمان ها می توانند در سرمایه گذاری قابل توجهی که قبلاً برای مدیریت معماری داخلی انجام می دادند، صرفه جویی کنند.

Scalability: Scalability is one of the most attractive features of the serverless stack. It scales dynamically with increased traffic and also increases the number of concurrent executing functions. This is possible as operations are smaller and can be loaded into any of the free executors available. In contrast, this was difficult earlier as the whole application needed to be loaded on memory.

مقیاس پذیری: مقیاس پذیری یکی از جذاب ترین ویژگی های پشته بدون سرور است. با افزایش ترافیک به صورت پویا مقیاس می شود و همچنین تعداد عملکردهای همزمان را افزایش می دهد. این امکان پذیر است زیرا عملیات ها کوچکتر هستند و می توانند در هر یک از مجری های رایگان موجود بارگذاری شوند. در مقابل، این کار قبلاً دشوار بود زیرا کل برنامه باید روی حافظه بارگذاری شود.

Efficient usage: You pay only for the amount of computing time. So you are not charged a penny if there were zero usage in a month.

استفاده کارآمد: شما فقط برای مقدار زمان محاسبه پرداخت می کنید. بنابراین اگر در یک ماه استفاده صفر باشد، یک پنی از شما دریافت نمی شود.

**Candidates for migrating services into serverless architecture**  
  
Though serverless seems too appealing, not all applications can be run on serverless architecture. Here are some parameters which will help us select which applications can be used well on serverless architecture.

نامزدهای مهاجرت خدمات به معماری بدون سرور

اگرچه بدون سرور خیلی جذاب به نظر می رسد، اما همه برنامه ها را نمی توان با معماری بدون سرور اجرا کرد. در اینجا چند پارامتر وجود دارد که به ما کمک می کند انتخاب کنیم کدام برنامه ها می توانند به خوبی در معماری بدون سرور استفاده شوند.

Short running workloads: Functions that are small and are expected to run quickly can be deployed on serverless architecture. For example, an image processing application will be a good fit for serverless architecture. More computational power can be provided or modified for faster execution. There is no need for a separate image processing server.

بارهای کاری کوتاه مدت: عملکردهایی که کوچک هستند و انتظار می رود به سرعت اجرا شوند را می توان در معماری بدون سرور مستقر کرد. به عنوان مثال، یک برنامه پردازش تصویر برای معماری بدون سرور مناسب خواهد بود. قدرت محاسباتی بیشتری را می توان برای اجرای سریعتر ارائه کرد یا اصلاح کرد. نیازی به سرور پردازش تصویر جداگانه نیست.

Rest APIs: Basic CRUD applications can be built serverless. We can’t directly build web services around the serverless stack. However, we can create API endpoints and map them to functions. These functions get triggered when users hit these endpoints. We can also map responses that these functions return.

Rest API: برنامه‌های اصلی CRUD را می‌توان بدون سرور ساخت. ما نمی توانیم مستقیماً سرویس های وب را در اطراف پشته بدون سرور ایجاد کنیم. با این حال، ما می توانیم نقاط پایانی API را ایجاد کرده و آنها را به توابع نگاشت کنیم. این توابع زمانی فعال می شوند که کاربران به این نقاط پایانی برخورد کنند. همچنین می‌توانیم پاسخ‌هایی را که این توابع برمی‌گردانند ترسیم کنیم.

Periodic triggers: Schedulers or CRON jobs can be scheduled without a dedicated server. The schedulers are accurate and trigger functions that execute code at reliable intervals or a specific time. An organisation that needs to generate a daily report at a particular time at night will be a good example—or doing periodic health checks of the servers.

محرک‌های دوره‌ای: زمان‌بندی‌ها یا کارهای CRON را می‌توان بدون سرور اختصاصی برنامه‌ریزی کرد. زمانبندها دقیق هستند و توابع را راه اندازی می کنند که کد را در فواصل قابل اعتماد یا زمان مشخصی اجرا می کنند. سازمانی که نیاز به تهیه گزارش روزانه در یک زمان خاص در شب دارد، نمونه خوبی خواهد بود - یا انجام بررسی های سلامت دوره ای سرورها.

Communication: Cloud providers provide notification services that can deliver event-based notifications to other applications or messages to users via SMS, email, mobile push notifications. So applications that maintain their servers can do away and readily use these services.

ارتباطات: ارائه دهندگان ابر خدمات اعلان ارائه می کنند که می توانند اعلان های مبتنی بر رویداد را از طریق پیامک، ایمیل، اعلان های فشار موبایل به سایر برنامه ها یا پیام ها به کاربران ارائه دهند. بنابراین برنامه‌هایی که سرورهای خود را حفظ می‌کنند، می‌توانند از این سرویس‌ها استفاده کنند و به راحتی استفاده کنند.

Storage: Storage options allow developers to store documents as per demand. Hence developers can tweak their applications to store documents using these services rather than storing files on a dedicated server.

ذخیره سازی: گزینه های ذخیره سازی به توسعه دهندگان اجازه می دهد اسناد را بر اساس تقاضا ذخیره کنند. از این رو توسعه دهندگان می توانند برنامه های خود را برای ذخیره اسناد با استفاده از این سرویس ها به جای ذخیره فایل ها در سرور اختصاصی تغییر دهند.

**Drawbacks of EDA:**

Over-engineering: Sometimes, a straightforward API call between two services can serve the purpose, but if we are going with EDA, we require good infrastructure on both the ends (producer and consumer), i.e. queuing system, which will add additional costs. Secondly, instead of just focusing on business logic, the team also has an extra effort of reading events and validating the same.

Inconsistent behaviour: Update to the same event and duplication of an event makes the system more challenging to handle, making the system complex. This may also result in increased time for testing and debugging scenarios.

Error handling and troubleshooting: A typical application will generally have several message producers and consumers. Third-party tools might be required to be installed & configured to monitor the event flow effectively.

مهندسی بیش از حد: گاهی اوقات، یک تماس مستقیم API بین دو سرویس می‌تواند به این هدف عمل کند، اما اگر با EDA پیش برویم، به زیرساخت خوبی در هر دو طرف (تولیدکننده و مصرف‌کننده)، یعنی سیستم نوبت دهی نیاز داریم که هزینه‌های اضافی را اضافه می‌کند. ثانیاً، تیم به جای تمرکز بر منطق تجاری، تلاش بیشتری برای خواندن رویدادها و اعتبار بخشیدن به آن دارد.

رفتار ناسازگار: به روز رسانی به یک رویداد و تکرار یک رویداد، سیستم را چالش برانگیزتر می کند و سیستم را پیچیده می کند. این همچنین ممکن است منجر به افزایش زمان برای سناریوهای آزمایش و اشکال زدایی شود.

مدیریت خطا و عیب یابی: یک برنامه معمولی معمولا چندین تولید کننده و مصرف کننده پیام دارد. ممکن است نیاز به نصب و پیکربندی ابزارهای شخص ثالث برای نظارت بر جریان رویداد باشد.