**Event-driven architecture** (**EDA**) is a [software architecture](https://en.wikipedia.org/wiki/Software_architecture) paradigm promoting the production, detection, consumption of, and reaction to [events](https://en.wikipedia.org/wiki/Event_(computing)).

(EDA) یک الگوی معماری نرم افزاری است که باعث تولید ، کشف ، مصرف و واکنش به حوادث می شود.

An *event* can be defined as "a significant change in [state](https://en.wikipedia.org/wiki/State_(computer_science))".[[1]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-1) For example, when a consumer purchases a car, the car's state changes from "for sale" to "sold". A car dealer's system architecture may treat this state change as an event whose occurrence can be made known to other applications within the architecture. From a formal perspective, what is produced, published, propagated, detected or consumed is a (typically asynchronous) message called the event notification, and not the event itself, which is the state change that triggered the message emission. Events do not travel, they just occur. However, the term *event* is often used [metonymically](https://en.wikipedia.org/wiki/Metonymy) to denote the notification message itself, which may lead to some confusion. This is due to Event-Driven architectures often being designed atop **message-driven architectures**, where such communication pattern requires one of the inputs to be text-only, the message, to differentiate how each communication should be handled.

یک رویداد را می توان به عنوان "تغییر مهمی در وضعیت" تعریف کرد. [1] به عنوان مثال ، هنگامی که یک مشتری یک اتومبیل خریداری می کند ، وضعیت خودرو از "برای فروش" به "فروخته شده" تغییر می کند. معماری سیستم فروشندگی ماشین ممکن است این تغییر حالت را به عنوان رویدادی که وقوع آنرا می توان در سایر برنامه های داخل معماری مشخص کرد ، درمان کند. از دیدگاه رسمی ، آنچه تولید می شود ، منتشر می شود ، تبلیغ می شود ، شناسایی می شود یا مصرف می شود ، پیام (معمولاً ناهمزمان) به نام اطلاع رسانی رویداد است و نه خود رویداد ، آن تغییر حالت است که باعث انتشار پیام می شود. رویدادها سفر نمی کنند ، فقط اتفاق می افتند. با این حال ، اصطلاح رویداد اغلب به صورت مترادفانه برای بیان پیام پیام اعلان استفاده می شود ، که ممکن است منجر به سردرگمی شود. این امر به دلیل معماری رویدادهای محور است که اغلب در بالای معماریهای پیام محور طراحی می شوند ، که در آن الگوی ارتباطی برای تمایز چگونگی انجام هرگونه ارتباط ، نیاز به یکی از ورودی ها به صورت متن است.

This [architectural pattern](https://en.wikipedia.org/wiki/Architectural_pattern) may be applied by the design and implementation of applications and systems that transmit events among [loosely coupled software components](https://en.wikipedia.org/wiki/Loose_coupling) and [services](https://en.wikipedia.org/wiki/Service_(systems_architecture)). An event-driven system typically consists of event emitters (or agents), event consumers (or sinks), and event channels. Emitters have the responsibility to detect, gather, and transfer events. An Event Emitter does not know the consumers of the event, it does not even know if a consumer exists, and in case it exists, it does not know how the event is used or further processed. Sinks have the responsibility of applying a reaction as soon as an event is presented. The reaction might or might not be completely provided by the sink itself. For instance, the sink might just have the responsibility to filter, transform and forward the event to another component or it might provide a self-contained reaction to such event. Event channels are conduits in which events are transmitted from event emitters to event consumers. The knowledge of the correct distribution of events is exclusively present within the event channel.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)] The physical implementation of event channels can be based on traditional components such as [message-oriented middleware](https://en.wikipedia.org/wiki/Message-oriented_middleware) or point-to-point communication which might require a more appropriate transactional executive framework[[*clarify*](https://en.wikipedia.org/wiki/Wikipedia:Please_clarify)].

این الگوی معماری ممکن است با طراحی و اجرای برنامه ها و سیستم هایی که حوادث را در بین اجزای نرم افزاری و خدمات بی سیم همراه انتقال می دهند ، اعمال شود. یک سیستم مبتنی بر رویداد معمولاً از پخش کننده های رویداد (یا عوامل) ، مصرف کنندگان رویداد (یا غرق) و کانال های رویداد تشکیل شده است. انتشار دهنده ها مسئولیت شناسایی ، جمع آوری و انتقال حوادث را بر عهده دارند. یک Event Emitter مصرف کنندگان این رویداد را نمی شناسد ، حتی نمی داند مصرف کننده وجود دارد یا در صورت وجود ، نمی داند که چگونه این رویداد مورد استفاده قرار می گیرد یا بیشتر پردازش می شود. سینک ها وظیفه دارند به محض ارائه یک رویداد واکنش نشان دهند. واکنش ممکن است یا ممکن است توسط خود سینک به طور کامل ارائه نشده باشد. به عنوان مثال ، سینک ظرفشویی فقط می تواند مسئولیت فیلترینگ ، تبدیل و انتقال رویداد را به مؤلفه دیگری داشته باشد یا ممکن است واکنشی به خودی خود برای چنین رویدادی ارائه دهد. کانال های رویداد ، مجاری هستند که در آن حوادث از ناشرهای رویداد به مصرف کنندگان رویداد منتقل می شوند. دانش مربوط به توزیع صحیح رویدادها منحصراً در کانال رویداد وجود دارد. چارچوب اجرایی مناسب تراکنشی.

Building systems around an event-driven architecture simplifies horizontal scalability in [distributed computing](https://en.wikipedia.org/wiki/Distributed_computing) models and makes them more resilient to failure. This is because application state can be copied across multiple parallel snapshots for high-availability.[[2]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Fowler1-2) New events can be initiated anywhere, but more importantly propagate across the network of data stores updating each as they arrive. Adding extra nodes becomes trivial as well, you can simply take a copy of the application state, feed it a stream of events and run with it. [[3]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Fowler2-3)

سیستم های ساختمان سازی در اطراف یک معماری مبتنی بر رویداد ، مقیاس پذیری افقی را در مدل های محاسبات توزیع شده ساده تر کرده و آنها را در مقابل شکست مقاومت می کند. دلیل این است که حالت برنامه را می توان در چندین عکس موازی برای دسترسی زیاد کپی کرد. [2] رویدادهای جدید را می توان در هر نقطه آغاز کرد ، اما مهمتر از همه در شبکه فروشگاه های داده که هر زمان که می رسند ، به روز می شوند. اضافه کردن گره های اضافی نیز امری بی اهمیت است ، می توانید به سادگی یک نسخه از برنامه برنامه را بگیرید ، آن را به یک جریان از وقایع وصل کنید و با آن اجرا کنید.

Event-driven architecture can complement [service-oriented architecture](https://en.wikipedia.org/wiki/Service-oriented_architecture) (SOA) because services can be activated by triggers fired on incoming events.[[4]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Hanson1-4)[[5]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-5) This paradigm is particularly useful whenever the sink does not provide any self-contained executive[[*clarify*](https://en.wikipedia.org/wiki/Wikipedia:Please_clarify)].

معماری مبتنی بر رویداد می تواند معماری خدمات محور (SOA) را تکمیل کند زیرا خدمات را می توان با محرک های ناشی از وقایع ورودی فعال کرد. [4] [5] این الگوی به ویژه مفید است هر زمان که سینک هیچ اجرایی را در خود جای ندهد.

[SOA 2.0](https://en.wikipedia.org/wiki/Event-driven_SOA) evolves the implications SOA and EDA architectures provide to a richer, more robust level by leveraging previously unknown causal relationships to form a new event pattern.[[*vague*](https://en.wikipedia.org/wiki/Wikipedia:Vagueness)] This new [business intelligence](https://en.wikipedia.org/wiki/Business_intelligence) pattern triggers further autonomous human or automated processing that adds exponential value to the enterprise by injecting value-added information into the recognized pattern which could not have been achieved previously.[[*vague*](https://en.wikipedia.org/wiki/Wikipedia:Vagueness)]

SOA 2.0 مفاهیم SOA و EDA را با استفاده از روابط علّی قبلاً ناشناخته برای شکل دادن به یک الگوی رویداد جدید ، پیشرفت می کند و معماری های SOA و EDA را به سطح غنی تر و محکمی تری ارائه می دهد. این شرکت با تزریق اطلاعات با ارزش افزوده به الگوی شناخته شده ای که قبلاً نمی توان به دست آورد.

**Event structure**

An event can be made of two parts, the event header and the event body. The event header might include information such as event name, time stamp for the event, and type of event. The event body provides the details of the state change detected. An event body should not be confused with the pattern or the logic that may be applied in reaction to the occurrence of the event itself. [CloudEvents](https://cloudevents.io/) provides an Open Source specification for describing event data in a common way.

یک رویداد را می توان از دو بخش ، عنوان رویداد و بدن رویداد ایجاد کرد. عنوان رویداد ممکن است شامل اطلاعاتی مانند نام رویداد ، تمبر زمان رویداد و نوع رویداد باشد. بدنه رویداد جزئیات تغییر حالت کشف شده را ارائه می دهد. بدن یک رویداد نباید با الگو یا منطقی که ممکن است در واکنش به وقوع خود واقعه اعمال شود ، اشتباه گرفته شود. CloudEvents مشخصات منبع باز را برای توصیف داده های رویداد به روشی مشترک فراهم می کند.

**Event flow layers**

An event driven architecture may be built on four logical layers, starting with the sensing of an event (i.e., a significant temporal state or fact), proceeding to the creation of its technical representation in the form of an event structure and ending with a non-empty set of reactions to that event.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6)

یک معماری مبتنی بر رویداد ممکن است بر روی چهار لایه منطقی ساخته شود ، با شروع یک واقعه (یعنی یک وضعیت یا واقعیت زمانی قابل توجه) و شروع به ایجاد نمایندگی فنی آن در قالب یک ساختار رویداد و با پایان دادن به غیر مجموعه ای از واکنش ها به آن رویداد.

The first logical layer is the event generator, which senses a fact and represents that fact as an event message. As an example, an event generator could be an email client, an E-commerce system, a monitoring agent or some type of physical sensor.

اولین لایه منطقی تولید کننده رویداد است که یک واقعیت را حس می کند و آن واقعیت را به عنوان پیام رویداد نشان می دهد. به عنوان نمونه ، یک تولید کننده رویداد می تواند مشتری ایمیل ، یک سیستم تجارت الکترونیکی ، یک عامل نظارت یا نوعی حسگر فیزیکی باشد.

Converting the data collected from such a diverse set of data sources to a single standardized form of data for evaluation is a significant task in the design and implementation of this first logical layer.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6) However, considering that an event is a strongly declarative frame, any informational operations can be easily applied, thus eliminating the need for a high level of standardization.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

تبدیل داده های جمع آوری شده از چنین مجموعه متنوعی از منابع داده به یک فرم استاندارد و استاندارد برای ارزیابی ، یک کار مهم در طراحی و اجرای این اولین لایه منطقی است. [6] با این وجود ، با توجه به اینکه یک رویداد یک چارچوب کاملاً اعلانی است ، می توان به راحتی هرگونه عملیات اطلاع رسانی را انجام داد ، بنابراین نیاز به سطح بالایی از استاندارد سازی را از بین می برد.

### Event channel

This is the second logical layer. An event channel is a mechanism of propagating the information collected from an event generator to the event engine[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6) or sink. This could be a TCP/IP connection, or any type of an input file (flat, XML format, e-mail, etc.). Several event channels can be opened at the same time. Usually, because the event processing engine has to process them in near real time, the event channels will be read asynchronously. The events are stored in a queue, waiting to be processed later by the event processing engine.

این دومین لایه منطقی است. کانال رویداد مکانیسم انتشار اطلاعاتی است که از یک ژنراتور رویداد به موتور رویداد [6] یا سینک جمع آوری می شود. این می تواند یک اتصال TCP / IP یا هر نوع فایل ورودی (تخت ، فرمت XML ، ایمیل و غیره) باشد. چندین کانال رویداد را می توانید همزمان باز کنید. معمولاً به دلیل اینکه موتور پردازش رویداد باید آنها را در زمان واقعی تقریباً پردازش کند ، کانالهای رویداد بصورت غیر همزمان خوانده می شوند. این رویدادها در یک صف ذخیره می شوند و انتظار دارند که بعدا توسط موتور پردازش رویداد پردازش شود.

### Event processing engine

The event processing engine is the logical layer responsible for identifying an event, and then selecting and executing the appropriate reaction. It can also trigger a number of assertions. For example, if the event that comes into the event processing engine is a product ID low in stock, this may trigger reactions such as “Order product ID” and “Notify personnel”.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6)

موتور پردازش رویداد یک لایه منطقی است که وظیفه شناسایی یک رویداد و سپس انتخاب و اجرای واکنش مناسب را دارد. همچنین می تواند ادعاهای متعددی را برانگیزد. به عنوان مثال ، اگر رویدادی که وارد موتور پردازش رویداد می شود ، یک محصول کم موجود در سطح ID داشته باشد ، ممکن است واکنش هایی مانند "سفارش محصول شناسه سفارش" و "اطلاع به پرسنل" ایجاد کند.

### Downstream event-driven activity

This is the logical layer where the consequences of the event are shown. This can be done in many different ways and forms; e.g., an email is sent to someone and an application may display some kind of warning on the screen.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6) Depending on the level of automation provided by the sink (event processing engine) the downstream activity might not be required.

این لایه منطقی است که در آن پیامدهای رویداد نشان داده می شود. این می تواند به روش ها و اشکال مختلفی انجام شود. به عنوان مثال ، یک ایمیل به شخصی ارسال می شود و یک برنامه ممکن است نوعی اخطار را روی صفحه نمایش دهد. [6] بسته به سطح اتوماسیون ارائه شده توسط سینك (موتور پردازش رویداد) ، فعالیت پایین دستی لازم نیست.

## Event processing styles

There are three general styles of event processing: simple, stream, and complex. The three styles are often used together in a mature event-driven architecture.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6)

سه سبک کلی پردازش رویداد وجود دارد: ساده ، جریان و پیچیده. این سه سبک اغلب در یک معماری بالغ مبتنی بر رویداد استفاده می شوند.

### Simple event processing

Simple event processing concerns events that are directly related to specific, measurable changes of condition. In simple event processing, a notable event happens which initiates downstream action(s). Simple event processing is commonly used to drive the real-time flow of work, thereby reducing lag time and cost.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6)

پردازش رویداد ساده مربوط به اتفاقاتی است که با تغییرات خاص و قابل اندازه گیری شرایط ارتباط مستقیم دارند. در پردازش یک رویداد ساده ، یک رویداد قابل توجه اتفاق می افتد که اقدامات (های) پایین دست را آغاز می کند. پردازش رویداد ساده معمولاً برای هدایت جریان واقعی کار مورد استفاده قرار می گیرد و در نتیجه زمان و هزینه را کاهش می دهد.

For example, simple events can be created by a sensor detecting changes in tire pressures or ambient temperature. The car's tire incorrect pressure will generate a simple event from the sensor that will trigger a yellow light advising the driver about the state of a tire.

به عنوان مثال ، حوادث ساده می تواند توسط یک سنسور ایجاد شود که تغییرات فشار تایرها یا دمای محیط را تشخیص می دهد. فشار نادرست لاستیک خودرو باعث ایجاد یک اتفاق ساده از سنسور خواهد شد که باعث ایجاد یک چراغ زرد خواهد شد که به راننده در مورد وضعیت یک تایر توصیه می کند.

### Event stream processing

In [event stream processing](https://en.wikipedia.org/wiki/Event_stream_processing) (ESP), both ordinary and notable events happen. Ordinary events (orders, RFID transmissions) are screened for notability and streamed to information subscribers. Event stream processing is commonly used to drive the real-time flow of information in and around the enterprise, which enables in-time decision making.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6)

در پردازش جریان رویداد (ESP) ، هر دو رویداد عادی و قابل توجه اتفاق می افتد. رویدادهای معمولی (سفارشات ، انتقال RFID) به دلیل قابل توجه بودن نمایش داده می شوند و به مشترکین اطلاعات منتقل می شوند. پردازش جریان رویداد معمولاً برای هدایت جریان واقعی اطلاعات به داخل و اطراف شرکت مورد استفاده قرار می گیرد که تصمیم گیری در زمان را امکان پذیر می کند.

### Complex event processing

[Complex event processing](https://en.wikipedia.org/wiki/Complex_event_processing) (CEP) allows patterns of simple and ordinary events to be considered to infer that a complex event has occurred. Complex event processing evaluates a confluence of events and then takes action. The events (notable or ordinary) may cross event types and occur over a long period of time. The event correlation may be causal, temporal, or spatial. CEP requires the employment of sophisticated event interpreters, event pattern definition and matching, and correlation techniques. CEP is commonly used to detect and respond to business anomalies, threats, and opportunities.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6)

پردازش رویدادهای پیچیده (CEP) اجازه می دهد تا الگوهای رویدادهای ساده و معمولی در نظر بگیرند که یک رویداد پیچیده اتفاق افتاده است. پردازش رویدادهای پیچیده ، ترکیبی از وقایع را ارزیابی می کند و سپس اقدام می کند. این رویدادها (قابل توجه یا معمولی) ممکن است از انواع رویدادها عبور کنند و در مدت زمان طولانی رخ دهند. همبستگی این رویداد ممکن است علّی ، موقتی یا مکانی باشد. CEP به استخدام مفسران رویداد پیچیده ، تعریف الگوی رویداد و تطبیق ، و تکنیک های همبستگی نیاز دارد. CEP معمولاً برای تشخیص و پاسخ به ناهنجاریهای تجاری ، تهدیدات و فرصتها استفاده می شود.

### Online event processing

Online event processing (OLEP) uses asynchronous distributed event-logs to process complex events and manage persistent data[[7]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-7). OLEP allows to reliably compose related events of a complex scenario across heterogeneous systems. It therewith enables very flexible distribution patterns with high scalability and offers strong consistency. However, it cannot guarantee an upper bound to the processing time.

پردازش رویدادهای آنلاین (OLEP) برای پردازش رویدادهای پیچیده و مدیریت داده های مداوم از نگاشتهای رویدادهای توزیع شده ناهمزمان استفاده می کند [7]. OLEP اجازه می دهد تا رویدادهای مرتبط با سناریوی پیچیده را در سیستمهای ناهمگن تلفیق کند. از این طریق الگوهای توزیع بسیار انعطاف پذیر با مقیاس پذیری بالا را قادر می سازد و قوام قوی را ارائه می دهد. با این حال ، این نمی تواند یک حد بالایی با زمان پردازش را تضمین کند.

## Extreme loose coupling and well distributed

An event driven architecture is extremely loosely coupled and well distributed. The great distribution of this architecture exists because an event can be almost anything and exist almost anywhere. The architecture is extremely loosely coupled because the event itself doesn’t know about the consequences of its cause. e.g. If we have an alarm system that records information when the front door opens, the door itself doesn’t know that the alarm system will add information when the door opens, just that the door has been opened.[[6]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-Michelson1-6)

معماری محور رویداد کاملاً کاملاً همراه و توزیع خوبی است. توزیع بسیار زیاد این معماری به این دلیل وجود دارد که یک رویداد تقریباً می تواند هر چیزی باشد و تقریباً در هر کجا وجود داشته باشد. این معماری کاملاً کاملاً جفت شده است زیرا خود این رویداد از پیامدهای علت آن آگاهی ندارد. به عنوان مثال، اگر ما یک سیستم هشدار دهنده داشته باشیم که هنگام باز شدن درب جلو ، اطلاعات را ضبط می کند ، خود درب نمی داند که سیستم زنگ هشدار با باز شدن درب اطلاعات را اضافه می کند ، فقط با باز شدن درب.

### Semantic Coupling and further research

Event driven architectures have loose coupling within space, time and synchronization, providing a scalable infrastructure for information exchange and distributed workflows. However, event-architectures are tightly coupled, via event subscriptions and patterns, to the semantics of the underlying event schema and values. The high degree of semantic heterogeneity of events in large and open deployments such as smart cities and the sensor web makes it difficult to develop and maintain event-based systems. In order to address semantic coupling within event-based systems the use of approximate [semantic matching](https://en.wikipedia.org/wiki/Semantic_matching) of events is an active area of research.[[8]](https://en.wikipedia.org/wiki/Event-driven_architecture#cite_note-8)

معماریهای محور رویداد ، اتصال را در فضا ، زمان و هماهنگ سازی سست می کنند ، یک زیرساخت مقیاس پذیر را برای تبادل اطلاعات و گردش کار توزیع می کنند. با این حال ، معماری رویدادها ، از طریق اشتراک های رویدادها و الگوهای ، به معنای معیار برنامه و ارزش های رویداد اصلی محکم هستند. درجه بالای ناهمگونی معنایی وقایع در استقرارهای بزرگ و باز مانند شهرهای هوشمند و شبکه حسگر ، توسعه و نگهداری سیستمهای مبتنی بر رویداد را دشوار می کند. به منظور پرداختن به جفت معنایی در سیستمهای مبتنی بر رویداد ، استفاده از تطبیق معنایی تقریبی وقایع ، منطقه فعال تحقیق است.

## Implementations and examples

### Java Swing

The [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) [Swing](https://en.wikipedia.org/wiki/Swing_(Java)) API is based on an event driven architecture. This works particularly well with the motivation behind Swing to provide user interface related components and functionality. The API uses a nomenclature convention (e.g. "ActionListener" and "ActionEvent") to relate and organize event concerns. A class which needs to be aware of a particular event simply implements the appropriate listener, overrides the inherited methods, and is then added to the object that fires the event. A very simple example could be:

public class FooPanel extends JPanel implements ActionListener {

public FooPanel() {

super();

JButton btn = new JButton("Click Me!");

btn.addActionListener(this);

this.add(btn);

}

@Override

public void actionPerformed(ActionEvent ae) {

System.out.println("Button has been clicked!");

}

}

Alternatively, another implementation choice is to add the listener to the object as an [anonymous class](https://en.wikipedia.org/wiki/Class_(computer_programming)#Named_versus_anonymous). Below is an example.

public class FooPanel extends JPanel {

public FooPanel() {

super();

JButton btn = new JButton("Click Me!");

btn.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent ae) {

System.out.println("Button has been clicked!");

}

});

this.add(btn);

}

}

### JavaScript

(() => {

'use strict';

class EventEmitter {

constructor() {

this.events = new Map();

}

on(event, listener) {

if (typeof listener !== 'function') {

throw new TypeError('The listener must be a function');

}

let listeners = this.events.get(event);

if (!listeners) {

listeners = new Set();

this.events.set(event, listeners);

}

listeners.add(listener);

return this;

}

off(event, listener) {

if (!arguments.length) {

this.events.clear();

} else if (arguments.length === 1) {

this.events.delete(event);

} else {

const listeners = this.events.get(event);

if (listeners) {

listeners.delete(listener);

}

}

return this;

}

emit(event, ...args) {

const listeners = this.events.get(event);

if (listeners) {

for (let listener of listeners) {

listener.apply(this, args);

}

}

return this;

}

}

this.EventEmitter = EventEmitter;

})();

Usage:

const events = new EventEmitter();

events.on('foo', () => { console.log('foo'); });

events.emit('foo'); // Prints "foo"

events.off('foo');

events.emit('foo'); // Nothing will happen