-

-

-

Event-driven architectures are particularly **popular for microservices-based software applications**.

[**Microservices**](https://hazelcast.com/glossary/microservices/) **are built to perform very specific tasks, and these tasks are often based on the occurrence of some event. Therefore, event-driven architectures often form the backbone of microservices.**

-

-

-

# What is Event-driven Architecture?

**Event-driven architecture** (**EDA**) is a software **design pattern** that enables an organization to detect “events” or important business moments (such as a **transaction, site visit, shopping cart abandonment**, etc) and act on them in real time or near real time.

**This pattern replaces the traditional “request/response” architecture where services would have to wait for a reply before they could move onto the next task**. **The flow of event-driven architecture is run by events** and it is designed to respond to them or carry out some action in response to an event.

**Event-driven architecture is often referred to as “asynchronous**” communication. This means that **the sender and recipient** **don’t have to wait for each other to move onto their next task**. **Systems are not dependent on that one message. For instance, a telephone call is considered synchronous or more along the lines of the traditional “request/response” architecture**. Someone calls you and asks you to do something, the requestor waits while the responder completes the task, and then both parties hang up. **An asynchronous example would be text messaging**. **You send a message and in some cases, you don't even know who you are sending it to or if anyone’s listening, but you are not waiting for a response**.

## The Evolution of Event-driven Architecture

Over the past few years, there has been a movement from focusing on data at rest (service oriented architecture) to focusing on events (event-driven architecture). We are moving from accumulating data and data lakes to focus on data in flight and keeping track of it while it's moving from place to place. Traditionally, most systems operate in what you could think of as the data centric model where the data is the source of truth. The shift to event-driven architecture means moving from a data centric model to an event centric model. In the event-driven model, data is still important, but the events become the most important component. Whereas in the service oriented model, the highest priority was to make sure you don't lose any data. With event-driven architecture, the priority is to make sure you respond to events as they happen. Because there's a law of diminishing returns when it comes to events, the older they get, the less valuable they are. However, today, service oriented architecture and event-driven architecture are often used together.

Event-driven architecture often uses a log analogy of keeping track of things. Analysts talk about events as immutable things that happened. And if you want to figure out what happened in the past, you can go back and replay the log. Whereas in the data centric model, you're focused mainly on the state of the data as it is right now. And then the last analogy analysts use when describing the difference between data-centric and event-centric architectures is they often compare them between an information repository and a nervous system that carries messages around the enterprise.

When using an event-driven architecture you have event producers that generate and send event notifications and you may have one or more consumers of an event, where the receiving of the event triggers processing logic. For example, let's say Netflix has just uploaded a new movie. There could be several applications listening or waiting for that notification which then trigger their own internal systems to publish their own information about that event to their users. This differs from traditional request-reply messaging in that applications are still running and even though they may be listening for this event, they are not paralyzed while waiting for it. And, they are able to respond when the message is published. Thus, many services can be running in parallel.

## What is an Event?

An event is defined as a change of state of some key business system. For instance, somebody buys a product, someone else checks in for a flight or a bus is late arriving somewhere. And if one thinks about it, events exist everywhere and are constantly happening, no matter what industry. They are pervasive across any business. This includes **anything that creates a message by being produced, published, detected, or consumed is considered an event**. The event is separate from the message, because while the event is the occurrence, the message is the traveling notification that relays the occurrence. In event-driven architecture, an event will likely trigger one or more actions or processes in response to its occurrence. An example of an event might include:

* Request to reset a password
* A package arrived was delivered to its destination
* A grocery warehouse updates its inventory
* An unauthorized access attempt was denied

Each of these events is likely to trigger one or more actions or processes in response. One response might be simply to log the event for monitoring purposes. Others might be:

* An email to reset the password is sent to the customer
* The sales ticket is closed
* An order for more lettuce (or whatever materials are running low) is placed
* An account is locked and security personnel are notified

With event-driven architecture, when an event notification is sent, the system captures that something happened like a change in state has occurred and waits to send the reply to whoever requests it, whenever they request it. The application that received that message can either respond or wait to respond until the change in state has occurred that it is waiting for.

Applications built around an event-driven architecture enable more agile, scalable, contextual, and responsive digital business applications.

## How does Event-driven architecture work?

The components of an event-driven architecture can include three parts: producer, consumer, broker. The broker can be optional, particularly when you have a single producer and a single consumer that are in direct communication with each other and the producer just sends the events to the consumer. An example would be a producer that is sending only to a database or data warehouse so the events are collected and stored for analysis. Most commonly in enterprises, you have multiple sources sending out all types of events with one or more consumers interested in some or all of those events.

Let’s look at an example. If you are a retailer, you might be collecting all of the purchases that are happening at all of your stores all over the world. You are feeding them into your event-driven architecture that is watching for fraud, sending them to a credit card processor or whatever actions need to happen next. For a manufacturer, you have all kinds of data coming off your equipment that is telling you facts like temperature and pressure so you can monitor these events in real time and take actions like predict failures or schedule maintenance, depending on what the data is telling you.

-

-

-

-

-

-

-

-

-

Event-driven architecture

*Connected to:*

[Business intelligence](https://www.wikiwand.com/en/Business_intelligence)[Event-driven SOA](https://www.wikiwand.com/en/Event-driven_SOA)[Distributed computing](https://www.wikiwand.com/en/Distributed_computing)

From Wikipedia, the free encyclopedia

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

-

-

-

-

# Pattern: Event-driven architecture

# This pattern has been deprecated and replaced by the [Saga pattern](https://microservices.io/patterns/data/saga.html).

## Context

You have applied the [Database per Service](https://microservices.io/patterns/data/database-per-service.html) pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services. For example, lets imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases the application cannot simply use a local ACID transaction.

## Problem

How to maintain data consistency across services?

## Forces

* 2PC is not an option

## Solution

Use an event-driven, eventually consistent approach. Each service publishes an event whenever it update its data. Other service subscribe to events. When an event is received, a service updates its data.

## Example

An e-commerce application that uses this approach would work as follows:

1. **The Order Service creates an Order in a pending state and publishes an OrderCreated event.**
2. **The Customer Service receives the event and attempts to reserve credit for that Order. It then publishes either a Credit Reserved event or a CreditLimitExceeded event.**
3. **The Order Service receives the event from the Customer Service and changes the state of the order to either approved or cancelled**

## Resulting context

This pattern has the following benefits:

* It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

* The programming model is more complex

There are also the following issues to address:

* In order to be reliable, an application must atomically update its database and publish an event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one the patterns listed below.

## Related patterns

* The [Database per Service pattern](https://microservices.io/patterns/data/database-per-service.html) creates the need for this pattern
* The following patterns are ways to atomically update state and publish events:
  + [Event sourcing](https://microservices.io/patterns/data/event-sourcing.html)
  + [Transactional Outbox](https://microservices.io/patterns/data/transactional-outbox.html)
  + [Database triggers](https://microservices.io/patterns/data/database-triggers.html)
  + [Transaction log tailing](https://microservices.io/patterns/data/transaction-log-tailing.html)

## See also

The article [Event-Driven Data Management for Microservices](https://www.nginx.com/blog/event-driven-data-management-microservices/) by @crichardson describes this pattern

-

-

An *event* can be defined as "a significant change in [state](https://www.wikiwand.com/en/State_(computer_science))".[[1]](https://www.wikiwand.com/en/Event-driven_architecture#citenote1) 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://www.wikiwand.com/en/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.

This [architectural pattern](https://www.wikiwand.com/en/Architectural_pattern) may be applied by the design and implementation of applications and systems that transmit events among [loosely coupled software components](https://www.wikiwand.com/en/Loose_coupling) and [services](https://www.wikiwand.com/en/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://www.wikiwand.com/en/Wikipedia:Citation_needed)] The physical implementation of event channels can be based on traditional components such as [message-oriented middleware](https://www.wikiwand.com/en/Message-oriented_middleware) or point-to-point communication which might require a more appropriate transactional executive framework[[*clarify*](https://www.wikiwand.com/en/Wikipedia:Please_clarify)].

Building systems around an event-driven architecture simplifies horizontal scalability in [distributed computing](https://www.wikiwand.com/en/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://www.wikiwand.com/en/Event-driven_architecture#citenoteFowler12) 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://www.wikiwand.com/en/Event-driven_architecture#citenoteFowler23)

Event-driven architecture can complement [service-oriented architecture](https://www.wikiwand.com/en/Service-oriented_architecture) (SOA) because services can be activated by triggers fired on incoming events.[[4]](https://www.wikiwand.com/en/Event-driven_architecture#citenoteHanson14)[[5]](https://www.wikiwand.com/en/Event-driven_architecture#citenote5) This paradigm is particularly useful whenever the sink does not provide any self-contained executive[[*clarify*](https://www.wikiwand.com/en/Wikipedia:Please_clarify)].

[SOA 2.0](https://www.wikiwand.com/en/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://www.wikiwand.com/en/Wikipedia:Vagueness)] This new [business intelligence](https://www.wikiwand.com/en/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://www.wikiwand.com/en/Wikipedia:Vagueness)]

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.

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16)

Event Producer

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

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16) 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://www.wikiwand.com/en/Wikipedia:Citation_needed)]

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16) 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.

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16)

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16) Depending on the level of automation provided by the sink (event processing engine) the downstream activity might not be required.

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16)

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16)

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.

Event stream processing

In [event stream processing](https://www.wikiwand.com/en/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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16)

Complex event processing

[Complex event processing](https://www.wikiwand.com/en/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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16)

Online event processing

Online event processing (OLEP) uses asynchronous distributed event-logs to process complex events and manage persistent data.[[7]](https://www.wikiwand.com/en/Event-driven_architecture#citenote7) 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.

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://www.wikiwand.com/en/Event-driven_architecture#citenoteMichelson16)

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://www.wikiwand.com/en/Semantic_matching) of events is an active area of research.[[8]](https://www.wikiwand.com/en/Event-driven_architecture#citenote8)

Implementations and examples

Java Swing

The [Java](https://www.wikiwand.com/en/Java_(programming_language)) [Swing](https://www.wikiwand.com/en/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://www.wikiwand.com/en/Class_(computer_programming)#Named_versus_anonymous) and thus use the lambda notation (since Java 1.8). Below is an example.

**public** **class** **FooPanel** **extends** **JPanel** {

**public** **FooPanel**() {

**super**();

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

btn.addActionListener(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**

Object Pascal

Events are one of the fundamental elements of the [Object Pascal](https://www.wikiwand.com/en/Object_Pascal) language. The uni-cast model (one-to-one) is used here, i.e. the sender sends information to only one recipient. This limitation has the advantage that it does not need a special event listener. The event itself is a pointer to a method in another object. If the pointer is not empty, when an event occurs, the event handler is called. Events are commonly used in classes that support GUI. This is not the only area of application for events, however. The following code is an example of using events:

**unit** MyCustomClass;

**interface**

**uses**

Classes;

**type**

*{definition of your own event}*

TAccidentEvent = **procedure**(Sender: TObject; **const** AValue: Integer) **of** **object**;

**TMyCustomObject** = **class**(TObject)

**private**

FData: Integer; *// an example of a simple field in a class*

FOnAccident: TAccidentEvent; *// event - reference to a method in some object*

FOnChange: TNotifyEvent; *// event - reference to a method in some object*

**procedure** **SetData**(Value: Integer); *// a method that sets the value of a field in the class*

**protected**

**procedure** **DoAccident**(**const** AValue: Integer); **virtual**; *// a method that generates an event based on your own definition*

**procedure** **DoChange**; *// a method that generates an event based on a definition from the VCL library*

**public**

**constructor** **Create**; **virtual**; *// class constructor*

**destructor** **Destroy**; **override**; *// class destructor*

**published**

**property** Data: TAccidentEvent **read** FData **write** SetData; *// declaration of a property in a class*

**property** OnAccident: TAccidentEvent **read** FOnAccident **write** FOnAccident; *// exposing the event outside the class*

**property** OnChange: TNotifyEvent **read** FOnChange **write** FOnChange; *// exposing the event outside the class*

**procedure** **MultiplyBy**(**const** AValue: Integer); *// a method that uses its own definition of the event*

**end**;

**implementation**

**constructor** **TMyCustomObject**.**Create**;

**begin**

FData := 0;

**end**;

**destructor** **TMyCustomObject**.**Destroy**;

**begin**

FData := 0;

**inherited** Destroy;

**end**;

**procedure** **TMyCustomObject**.**DoAccident**(**const** AValue: Integer);

**begin**

**if** Assigned(FOnAccident)

**then** FOnAccident(Self, AValue);

**end**;

**procedure** **TMyCustomObject**.**DoChange**;

**begin**

**if** Assigned(FOnChange)

**then** FOnChange(Self);

**end**;

**procedure** **TMyCustomObject**.**MultiplyBy**(**const** AValue: Integer);

**begin**

FData := FData \* AValue;

**if** FData > 1000000

**then** DoAccident(FData);

**end**;

**procedure** **TMyCustomObject**.**SetData**(Value: Integer);

**begin**

**if** FData <> Value

**then**

**begin**

FData := Value;

DoChange;

**end**;

**end**.

The created class can be used as follows:

**...**

procedure TMyForm.ShowCustomInfo(Sender: TObject);

begin

**if** Sender is TMyCustomObject

then ShowMessage('Data has changed.');

end;

procedure TMyForm.PerformAcident(Sender: TObject; const AValue: Integer);

begin

**if** Sender is TMyCustomObject

then ShowMessage('The data has exceeded 1000000! New value is: ' + AValue.ToString);

end;

**...**

{declaring a variable that is an object of the specified class}

var

LMyObject: TMyCustomObject;

**...**

{creation of the object}

LMyObject := TMyCustomObject.Create;

**...**

{assigning a methods to an events}

LMyObject.OnChange := MyForm.ShowCustomInfo;

LMyObject.OnAccident := MyForm.PerformAcident;

**...**

{removing an object when it is no longer needed}

LMyObject.Free;

**...**

-

-

-

-

-

-