WIKIPEDIA

Event-driven architecture

Event-driven architecture (**EDA**), is a software architecture pattern promoting the production, detection, consumption of, and reaction to events.

An *event* can be defined as "a significant change in state".[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 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 may be applied by the design and implementation of applications and systems that transmit events among loosely coupled software components and services. 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. The physical implementation of event channels can be based on traditional components such as message-oriented middleware or point-to-point communication which might require a more appropriate transactional executive framework.

Building systems around an event-driven architecture simplifies horizontal scalability in 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] 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]

Event-driven architecture can complement service-oriented architecture (SOA) because services can be activated by triggers fired on incoming events.[4][5] This paradigm is particularly useful whenever the sink does not provide any self-contained executive.

SOA 2.0 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. This new 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.

Contents

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]

Event generator

The first logical layer is the event generator, which senses a fact and represents that fact into an event. A fact can be almost anything that can be sensed and as such, so too can an event generator. As an example, an event generator 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] 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.

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] 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]

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] 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]

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]

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 (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]

Complex event processing

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]

Online event processing

Online event processing (OLEP) uses asynchronous distributed event-logs to process complex events and manage persistent data[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.

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]

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 of events is an active area of research.[8]

Implementations and examples

Java Swing

The Java Swing 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. 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
```

See also

- Event-driven programming
- Process Driven Messaging Service
- Service-oriented architecture
- Event-driven SOA
- Space-based architecture
- Complex event processing
- Event stream processing
- Event Processing Technical Society
- Staged event-driven architecture (SEDA)
- Reactor pattern
- Autonomous peripheral operation

Articles

- Article defining the differences between EDA and SOA: *How EDA extends SOA and why it is important (http://soa-eda.blogspot.com/2006/11/how-eda-extends-soa-and-why-it-is.html)* by Jack van Hoof.
- Real-world example of business events flowing in an SOA: *SOA, EDA, and CEP - a winning combo (http://www.udidahan.com/2008/11/01/soa-eda-and-cep-a-winning-combo/)* by Udi Dahan.
- Article describing the concept of event data: *Analytics for hackers, how to think about event data (https://keen.io/blog/53958349217/analytics-for-hackers-how-to-think-about-event-data)* by Michelle Wetzler. (404 error)

References

1. K. Mani Chandy Event-Driven Applications: Costs, Benefits and Design Approaches, *California Institute of Technology*, 2006

2. Martin Fowler, Event Sourcing (https://martinfowler.com/eaaDev/EventSourcing.html), December, 2005
3. Martin Fowler, Parallel Model (https://martinfowler.com/eaaDev/ParallelModel.html), December, 2005
4. Jeff Hanson, Event-driven services in SOA (http://www.javaworld.com/javaworld/jw-01-2005/jw-0131-soa.html),*Javaworld*, January 31, 2005
5. Carol Sliwa Event-driven architecture poised for wide adoption (http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,81133,00.html), *Computerworld*, May 12, 2003
6. Brenda M. Michelson, Event-Driven Architecture Overview, *Patricia Seybold Group*, February 2, 2006
7. "Online Event Processing - ACM Queue" (https://queue.acm.org/detail.cfm?id=3321612). *queue.acm.org*. Retrieved 2019-05-30.
8. Hasan, Souleiman, Sean O'Riain, and Edward Curry. 2012. "Approximate Semantic Matching of Heterogeneous Events." (http://www.edwardcurry.org/publications/Hasan_DEBS_2012.pdf) In 6th ACM International Conference on Distributed Event-Based Systems (DEBS 2012), 252–263. Berlin, Germany: ACM. "DOI" (https://dx.doi.org/10.1145/2335484.2335512).

External links

- Event-Driven Applications: Costs, Benefits and Design Approaches (http://infospheres.caltech.edu/sites/default/files/Event-Driven%20Applications%20-%20Costs,%20Benefits%20and%20Design%20Approaches.pdf)
- Industry website on Complex Event Processing & Real Time Intelligence (http://www.complexevents.com)
- Website for the Event Processing Technical Society (http://www.ep-ts.com)
- 5th Anniversary Edition: Event-Driven Architecture Overview, Brenda M. Michelson (http://www.elementallinks.com/2011/02/06/5th-anniversary-edition-event-driven-architecture-overview/)
- Complex Event Processing and Service Oriented Architecture (http://news.tmcnet.com/news/2006/08/18/1816129.htm)
- How EDA extends SOA and why it is important (http://soa-eda.blogspot.com/2006/11/how-eda-extends-soa-and-why-it-is.html) by Jack van Hoof
- How to implement event-driven architecture (with RabbitMQ specifics) (https://medium.com/@wrong.about/event-driven-architecture-implementation-140c51820845)