

Otros patrones arquitecturales

Diseño de Sistemas Software
Curso 2017/2018

Carlos Pérez Sancho



Universitat d'Alacant
Universidad de Alicante

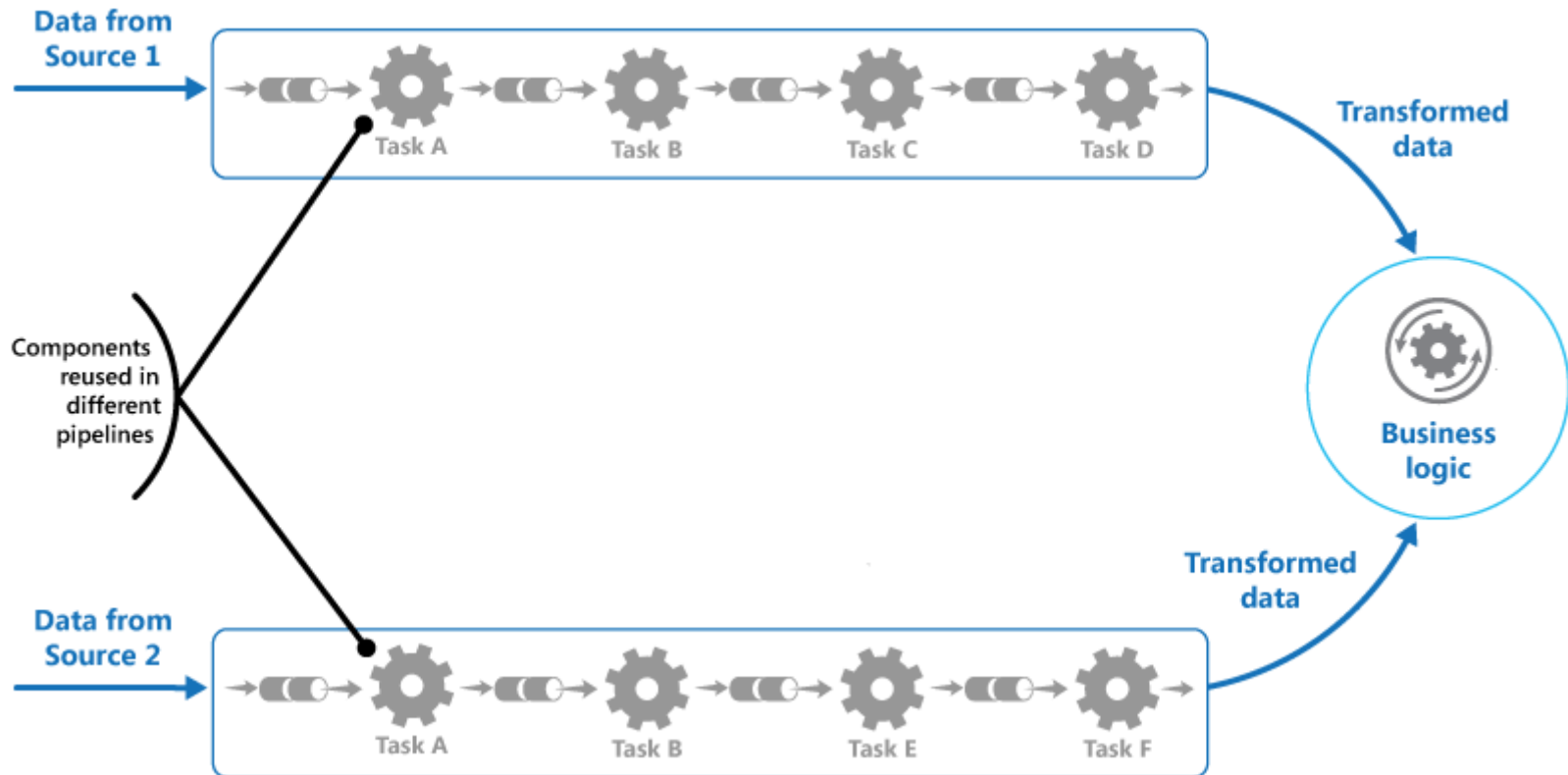
Departament de Llenguatges i Sistemes Informàtics
Departamento de Lenguajes y Sistemas Informáticos

Arquitectura tuberías y filtros

Patrones arquitecturales

Arquitectura tuberías y filtros

Los datos pasan por una serie de filtros, que transforman la información



Arquitectura tuberías y filtros

- Muy usado en sistema Unix, aunque no tiene por qué implementarse necesariamente mediante tuberías y línea de comandos
- Filosofía Unix
 - Modularidad
 - Simplicidad
 - Composición mediante el encadenamiento de programas sencillos para realizar tareas complejas
 - «Haz una cosa y hazla bien»

Arquitectura tuberías y filtros

- Ejemplo: Apertium

<https://www.apertium.org>



- Beneficios
 - Facilita el diagnóstico
 - Permite añadir fácilmente nuevos filtros entre dos módulos
 - Desarrollo de aplicaciones derivadas mediante la reutilización de módulos (interNOSTRUM → Apertium)

Arquitectura tuberías y filtros

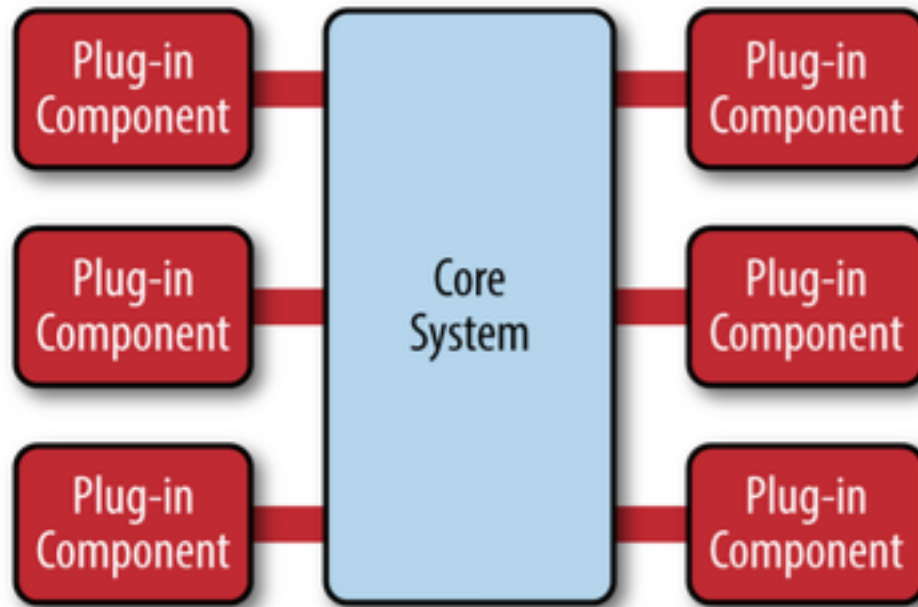
- Análisis del patrón
 - **Agilidad:** alta, debido a la modularidad
 - **Despliegue:** lento, normalmente se trata de aplicaciones que se ejecutan en las máquinas de los usuarios finales
 - **Pruebas:** fácil, se pueden probar los filtros de forma independiente
 - **Rendimiento:** alto, aunque depende de la complejidad de los filtros
 - **Escalabilidad:** baja, distribuir los filtros en distintos nodos añade complejidad
 - **Desarrollo:** fácil, si se usan los mecanismos de comunicación del sistema operativo (tuberías)

Arquitectura microkernel

Patrones arquitecturales

Arquitectura microkernel

También conocida como arquitectura de *plugins*, consiste en un núcleo central que se puede extender mediante módulos

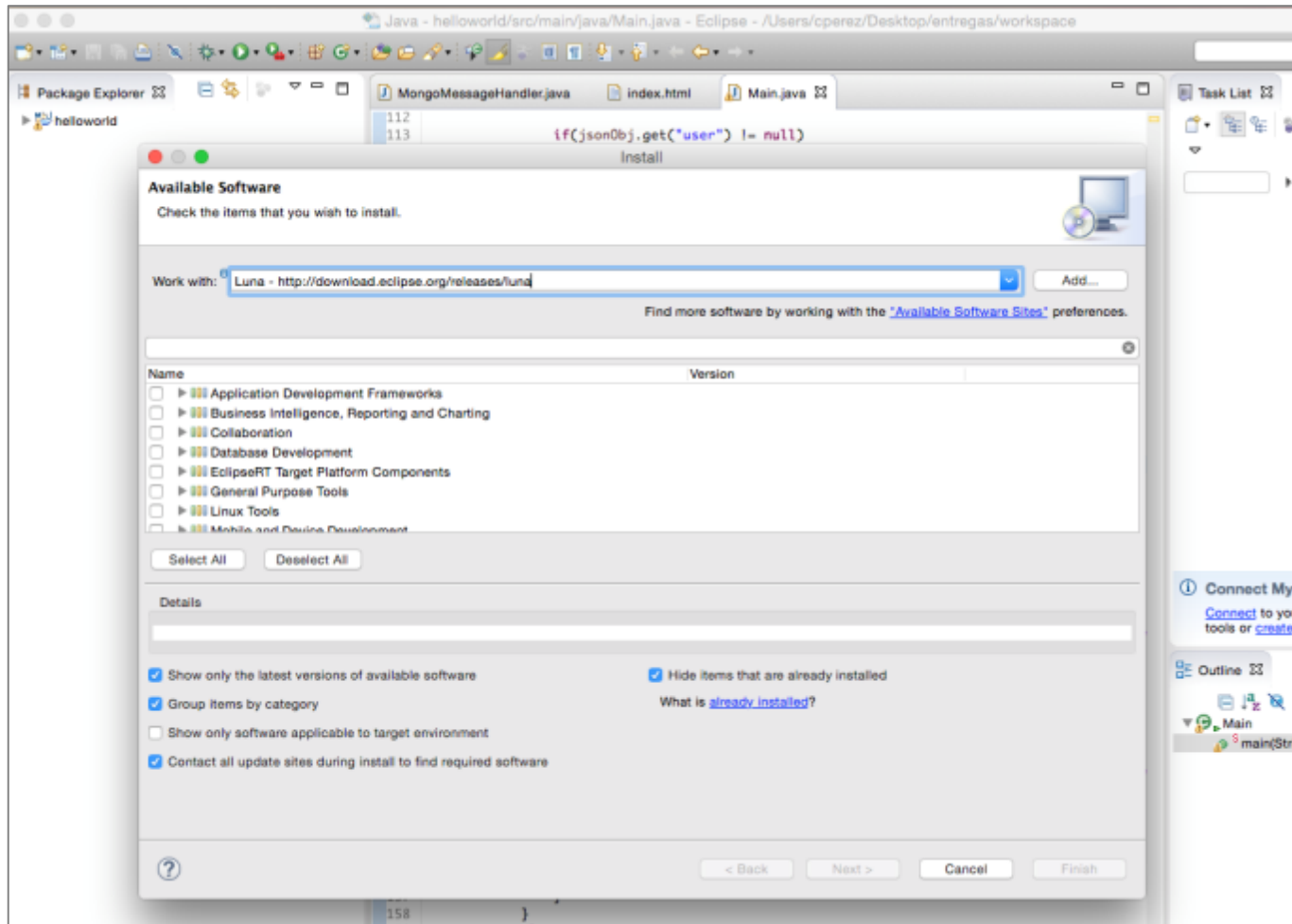


Arquitectura microkernel

- Componentes
 - **Núcleo central:** contiene la funcionalidad mínima para que el sistema funcione
 - ***Plugins***
 - Componentes independientes que añaden funcionalidades al núcleo central
 - Puede haber dependencias entre *plugins*
 - Se suelen organizar en un registro o repositorio para que el núcleo pueda obtener los *plugins* necesarios

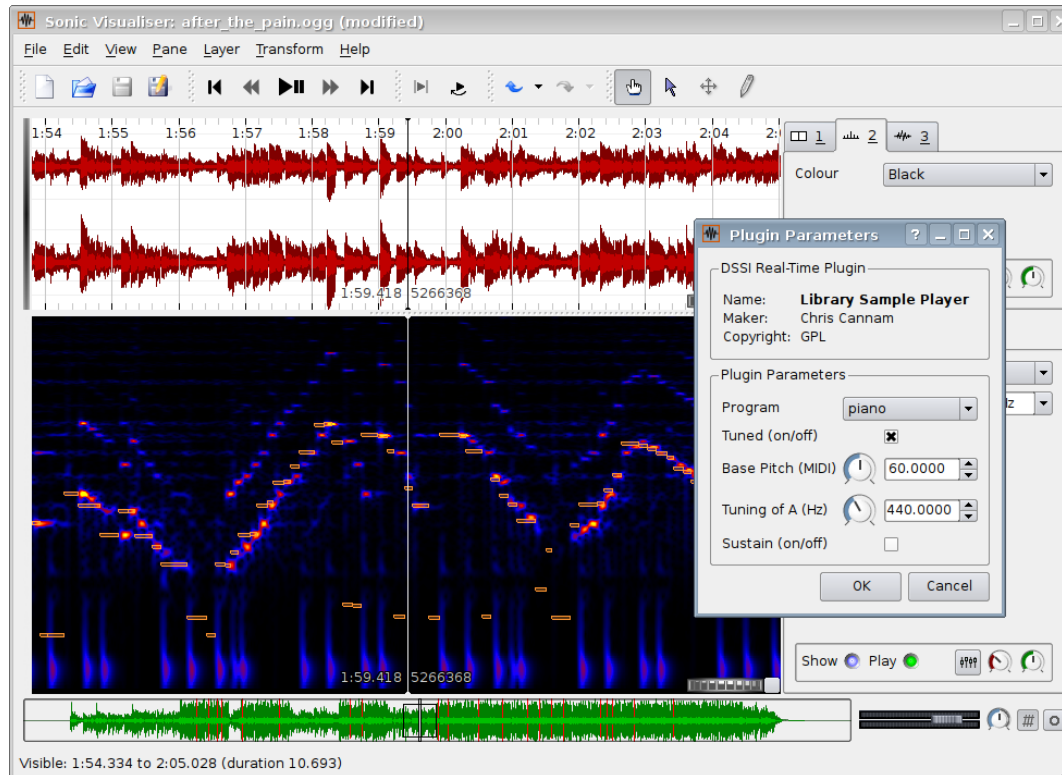
Arquitectura microkernel

Ejemplo: Eclipse IDE



Arquitectura microkernel

Ejemplo: Sonic Visualizer



<https://www.vamp-plugins.org/guide.pdf>

Arquitectura microkernel

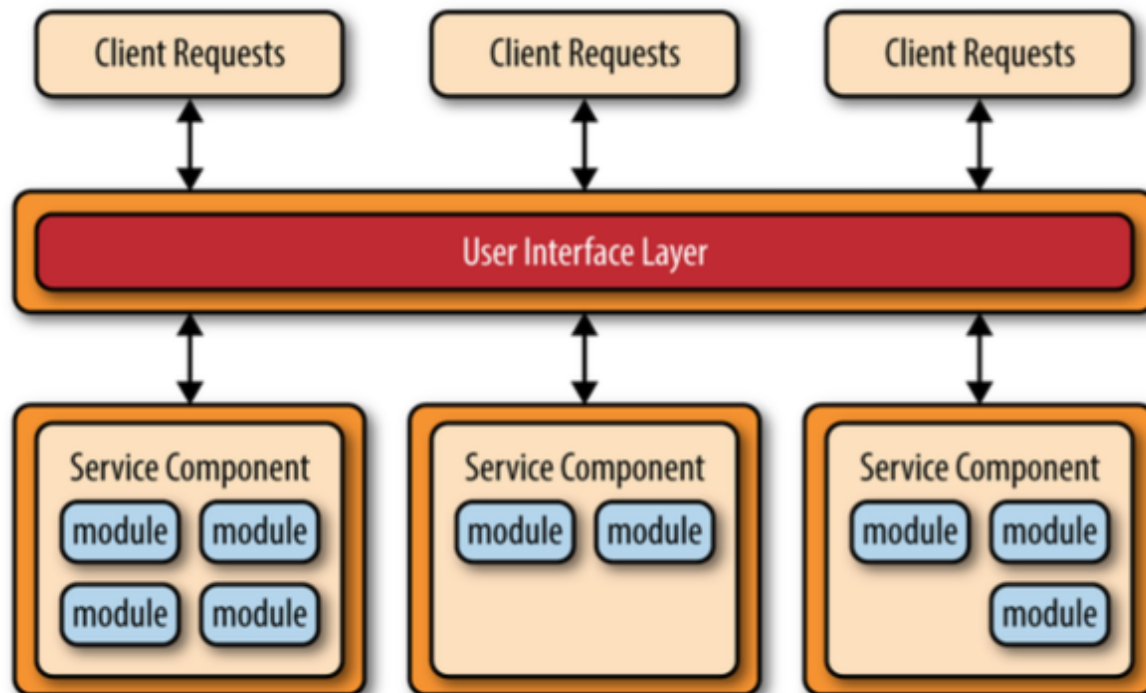
- Muy adecuado para el diseño y desarrollo evolutivo e incremental, y aplicaciones “producto”
- Análisis del patrón
 - **Agilidad:** alta, es sencillo añadir nuevos componentes
 - **Despliegue:** sencillo, se pueden añadir nuevos plugins en tiempo de ejecución
 - **Pruebas:** sencillo, se pueden probar los plugins por separado
 - **Rendimiento:** normalmente alto, ya que se pueden instalar únicamente los plugins necesarios
 - **Escalabilidad:** baja, normalmente diseñado como un único ejecutable
 - **Desarrollo:** difícil, el diseño del interfaz de los plugins debe planearse cuidadosamente. La gestión de versiones y repositorios de plugins añade complejidad

Arquitectura de microservicios

Patrones arquitecturales

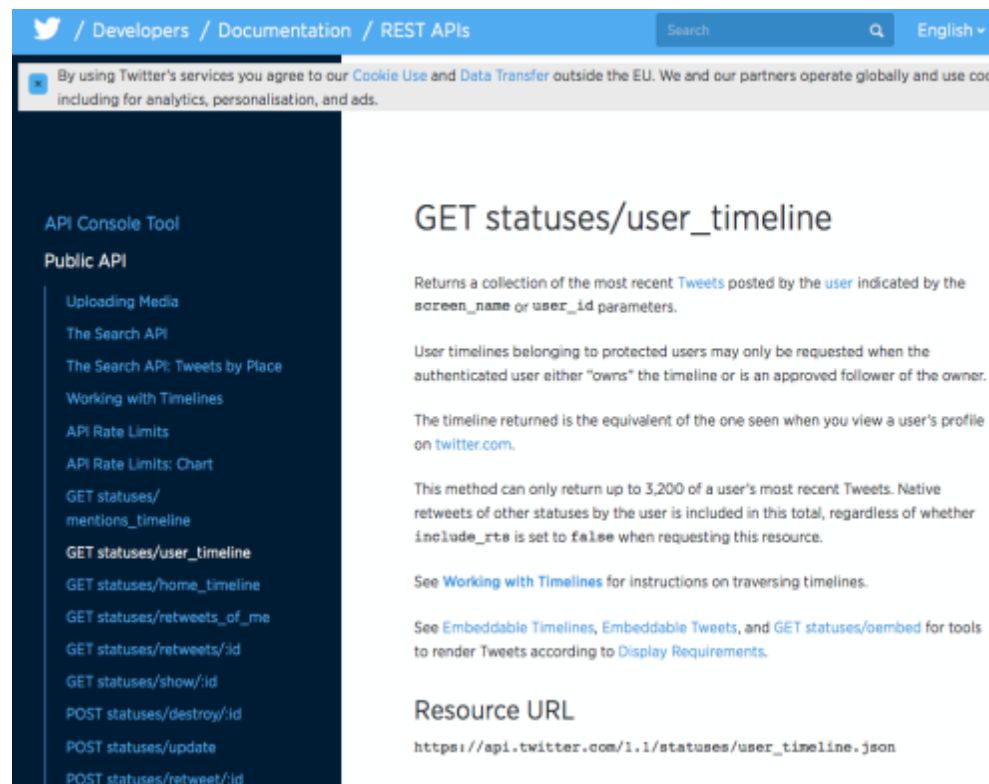
Arquitectura de microservicios

Arquitectura distribuida, el cliente se comunica con los servicios mediante algún protocolo de acceso remoto (REST, SOAP, RMI, etc.)



Arquitectura de microservicios

La implementación más frecuente utiliza HTTP como protocolo de comunicación, definiendo interfaces REST (REpresentational State Transfer) para acceder a los servicios



The screenshot shows the Twitter REST API documentation page for the `GET statuses/user_timeline` endpoint. The page has a blue header with the Twitter logo, navigation links for Developers, Documentation, and REST APIs, a search bar, and a language dropdown set to English. A cookie consent banner is visible below the header. On the left, a dark blue sidebar lists various API endpoints under the 'Public API' section, with `GET statuses/user_timeline` highlighted. The main content area has a white background and contains the following information:

GET statuses/user_timeline

Returns a collection of the most recent [Tweets](#) posted by the [user](#) indicated by the `screen_name` or `user_id` parameters.

User timelines belonging to protected users may only be requested when the authenticated user either "owns" the timeline or is an approved follower of the owner.

The timeline returned is the equivalent of the one seen when you view a user's profile on [twitter.com](#).

This method can only return up to 3,200 of a user's most recent Tweets. Native retweets of other statuses by the user is included in this total, regardless of whether `include_rts` is set to `false` when requesting this resource.

See [Working with Timelines](#) for instructions on traversing timelines.

See [Embeddable Timelines](#), [Embeddable Tweets](#), and [GET statuses/oembed](#) for tools to render Tweets according to [Display Requirements](#).

Resource URL

`https://api.twitter.com/1.1/statuses/user_timeline.json`

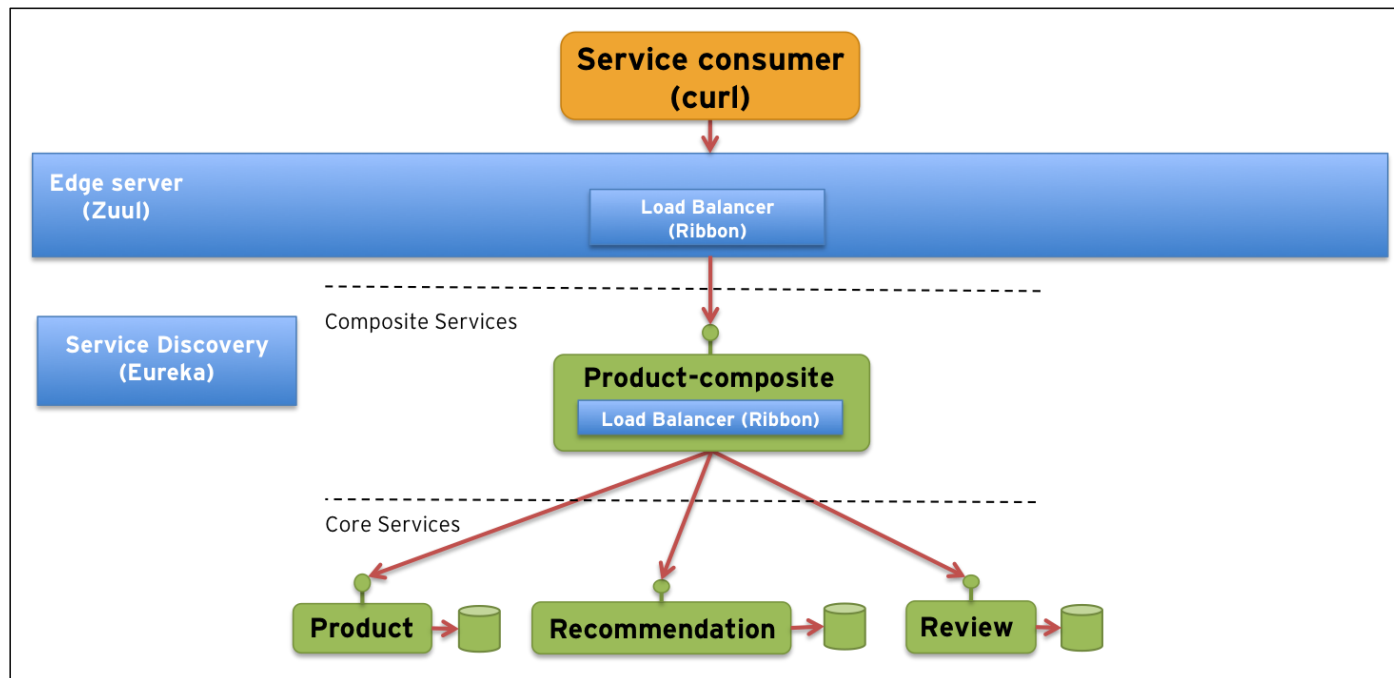
Arquitectura de microservicios

- Ejemplo: Netflix OSS

<https://netflix.github.io/>



- Aplicación de ejemplo



<http://callistaenterprise.se/blogg/teknik/2015/04/10/building-microservices-with-spring-cloud-and-netflix-oss-part-1/>

Arquitectura de microservicios

- Adecuada para el desarrollo de aplicaciones y servicios web
- Análisis del patrón
 - **Agilidad:** alta, los cambios afectan a componentes aislados
 - **Despliegue:** sencillo, favorece la integración continua
 - **Pruebas:** sencillo, debido a la independencia de los servicios
 - **Rendimiento:** bajo, debido a la naturaleza distribuida
 - **Escalabilidad:** alta, permite escalar los servicios por separado
 - **Desarrollo:** fácil, la independencia de los servicios reduce la necesidad de coordinación. El uso de protocolos de comunicación estándar facilita el desarrollo

Arquitectura orientada a eventos

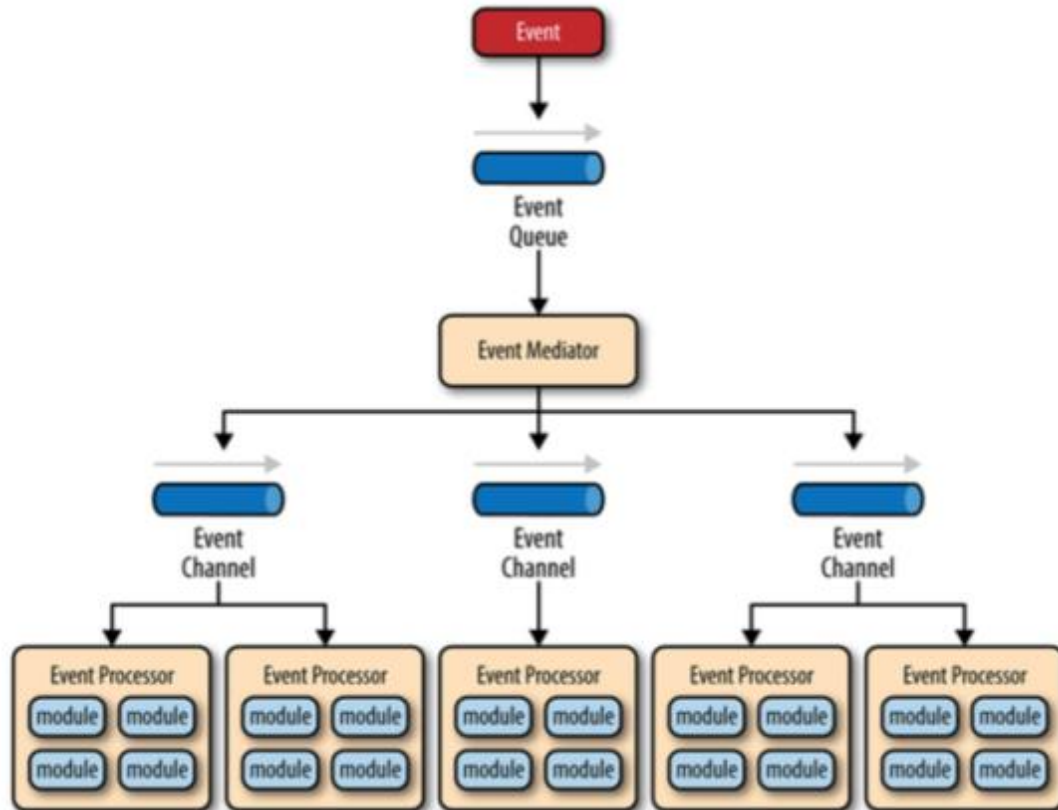
Patrones arquitecturales

Arquitectura orientada a eventos

- El sistema se compone de pequeños componentes que responden a eventos, y de algún mecanismo para gestionar las colas de eventos que se reciben
- Dos topologías alternativas
 - Mediador
 - Broker

Arquitectura orientada a eventos

Topología **mediador**: el procesamiento de eventos implica varios pasos que deben ejecutarse de manera orquestada

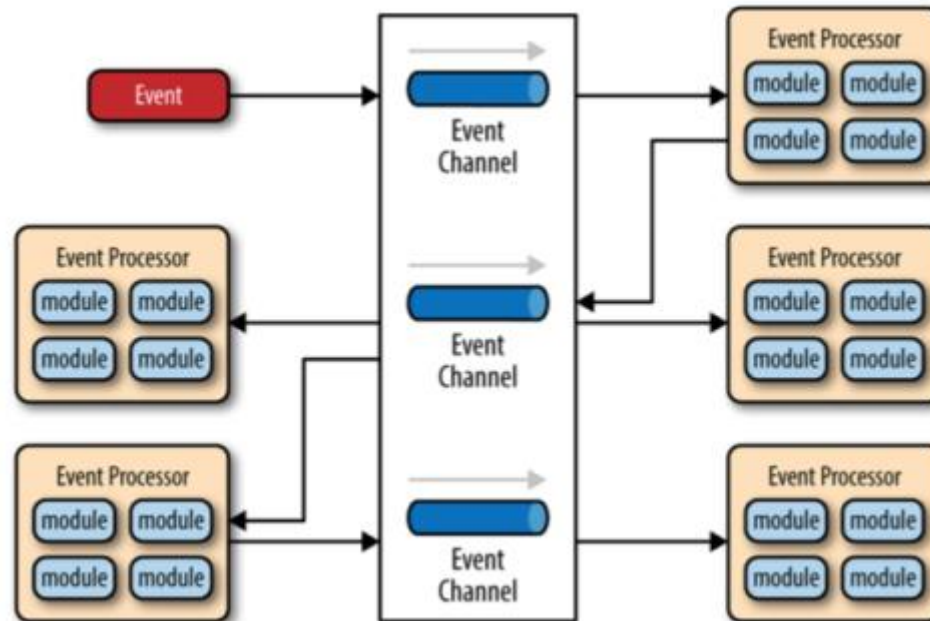


Arquitectura orientada a eventos

- Componentes
 - **Procesadores de eventos**
 - Contienen la lógica de negocio
 - Pequeñas unidades autocontenidas y altamente independientes del resto
 - **Mediador**
 - Recoge los eventos de inicio y los envía a los procesadores en el orden apropiado según el tipo de evento
 - Se puede implementar mediante soluciones open source, y definir usando lenguajes de definición de procesos (business process execution language, BPEL)

Arquitectura orientada a eventos

- Topología **broker**
 - Los procesadores de eventos se encadenan unos con otros mediante eventos que pasan a través del broker
 - Cuando un procesador de eventos termina su trabajo, genera un evento para que se ejecuten los siguientes componentes

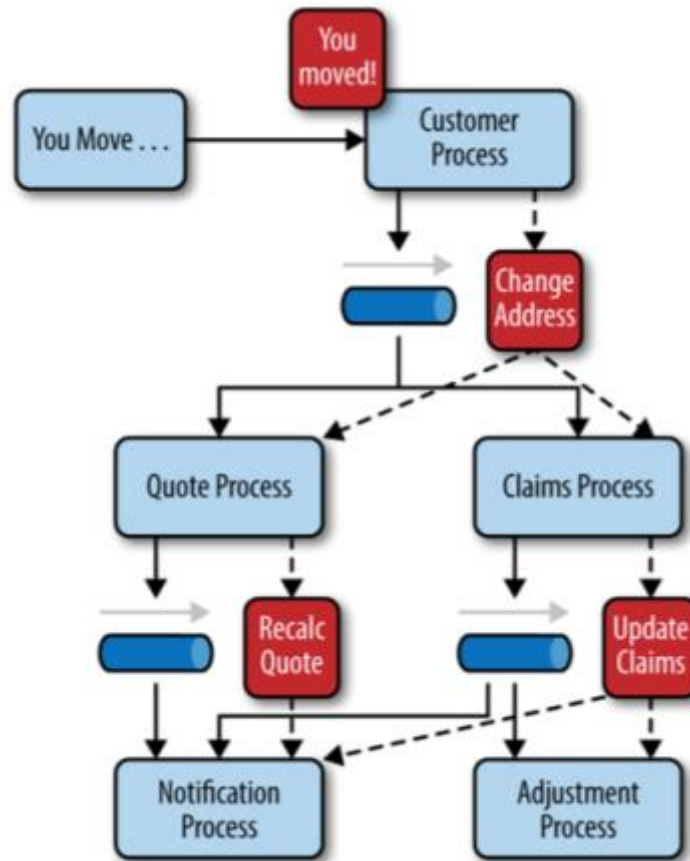


Arquitectura orientada a eventos

- Componentes
 - **Procesadores de eventos**
 - **Broker:** más ligero que el mediador, se encarga únicamente de gestionar las colas de eventos para que los procesadores no tengan que preocuparse de los detalles de implementación

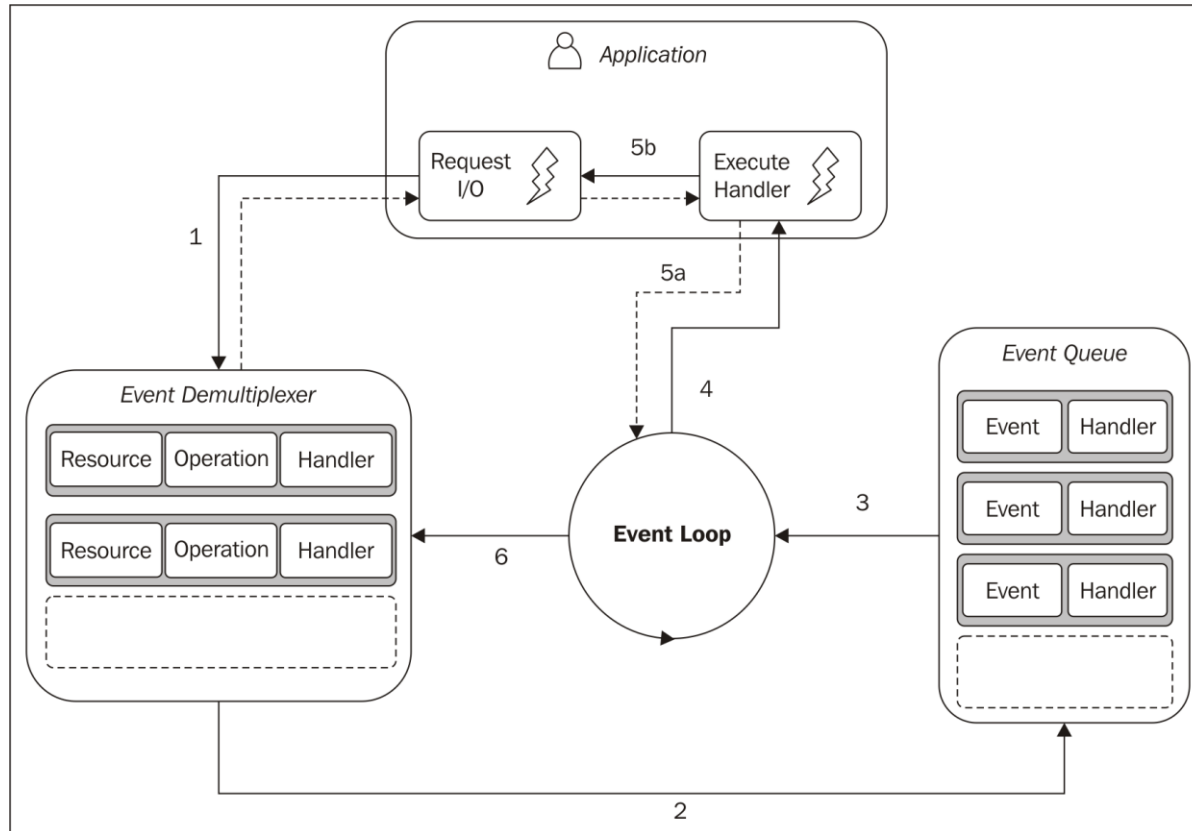
Arquitectura orientada a eventos

Ejemplo: un cliente de una aseguradora cambia de domicilio



Arquitectura orientada a eventos

Ejemplo: Node.js



This pattern provides the basis for the reactive programming paradigm

Arquitectura orientada a eventos

- Es una arquitectura compleja, de naturaleza asíncrona y distribuida
- Análisis del patrón
 - **Agilidad:** alta, los cambios normalmente afectan a uno o pocos componentes
 - **Despliegue:** sencillo, debido al bajo acoplamiento. Más complicado en el caso del mediador, ya que se debe actualizar cada vez que hay un cambio en los procesadores de eventos
 - **Pruebas:** complicado, debido a la naturaleza asíncrona y la necesidad de herramientas especializadas para generar eventos
 - **Rendimiento:** alto, debido a la posibilidad de paralelizar la ejecución de componentes
 - **Escalabilidad:** alta, los componentes pueden escalar por separado
 - **Desarrollo:** difícil, la gestión de errores es compleja

¿Preguntas?

- [Richards2015] Software Architecture Patterns. Mark Richards. O'Reilly, 2015