

Proyecto TMDAD

Contenido

Sistema	4
Patrones EIP	6
Patrones de canales	6
Canal punto a punto	6
Publicación-suscripción	6
Canal con tipo de datos	7
Adaptador de canal	7
Patrones de mensajes.....	7
Mensaje evento.....	7
Patrones de enrutado de mensajes.....	7
Filtro de mensajes.....	7
Patrones de transformación de mensajes	7
Enriquecedor de contenido.....	7
Filtro de contenido	7
Modelo arquitectural	8
Patrón repositorio.....	8
Puntos cumplidos.....	8
Aprobado.....	8
Notable	8
Sobresaliente.....	9
Plan de explotación.....	9
Máquinas virtuales.....	9
Google Cloud.....	9
Amazon Web Services.....	10
Microsoft Azure.....	11
RabbitMQ.....	11
MongoDB.....	12
Total.....	12
Diagramas.....	13
Diagrama de despliegue.....	13
Diagrama de conectores.....	14
Diagrama de componentes/módulos.....	15
Github	16

Sistema

El sistema permite de una manera más cómoda manejar la actividad en Twitter durante la realización de un hackathon. También permite realizar el registro en el mismo a través de Twitter.

Gracias a un simple tweet, los participantes pueden registrarse para participar en el evento. Para hacerlo, simplemente tendrán que realizar un tweet con la palabra clave del hackathon seguido de la palabra "registro" y después un conjunto de datos en el siguiente formato:

NombreDato:dato1;NombreDato2:dato2;NombreDato3:dato3

Por ejemplo:


#uCode registro nombre:Jorge;talla:M;edad:10

El sistema tiene dos *TweetProcessors*.

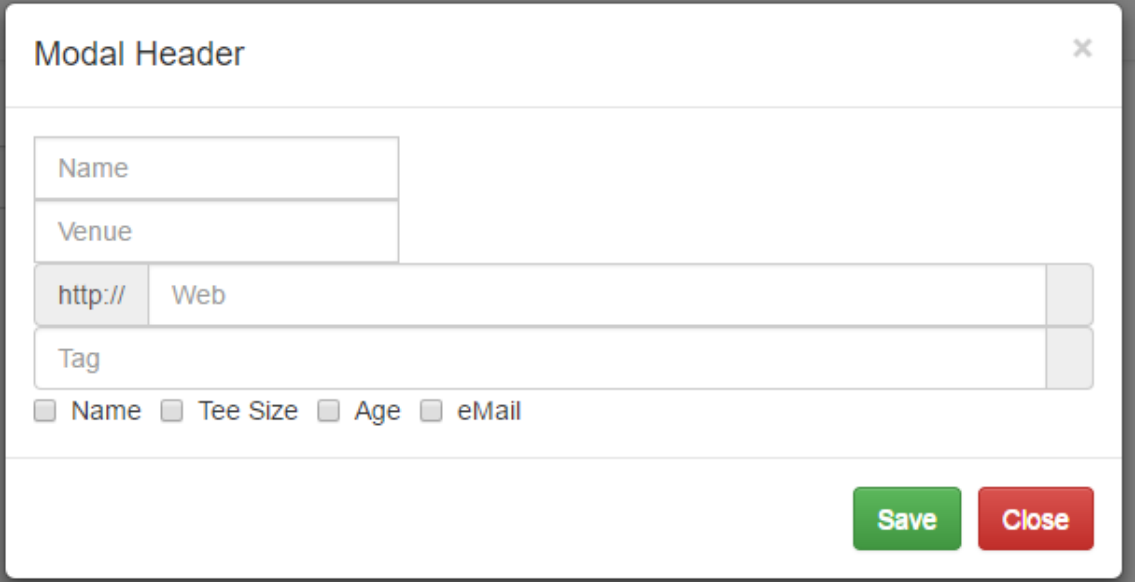
En la interfaz se puede seleccionar el hackathon sobre el cual se desea obtener información. Para cada hackathon aparecerán tres columnas. En la primera el conjunto de tweets que están relacionados con el hackathon.

En la segunda columna se puede ver un listado en tiempo real de los participantes que se han registrado en el evento a través del sistema de registro a través de Twitter.

En la tercera columna se puede ver el listado ordenado de Hashtags que más aparecen en los tweets relacionados con el evento.

El sistema permite la creación de hackathones. Para ello solo hay que presionar el botón de creación indicado con este símbolo .

Al hacerlo se despliega un formulario para la introducción de los datos del evento.



Modal Header

Name

Venue

http:// Web

Tag

☐ Name ☐ Tee Size ☐ Age ☐ eMail

Save Close

También se permite la edición de los hackathones existentes mediante el botón .

Patrones EIP

La utilización de patrones EIP facilita la creación de aplicaciones potentes, útiles y bien estructuradas. Facilita la creación de sistemas escalables debido al uso de pautas de diseño proporcionando soluciones sencillas a problemas comunes.

Los dos patrones más importantes en este sistema son el patrón repositorio y la arquitectura dirigida por eventos y varios otros dentro de este último.

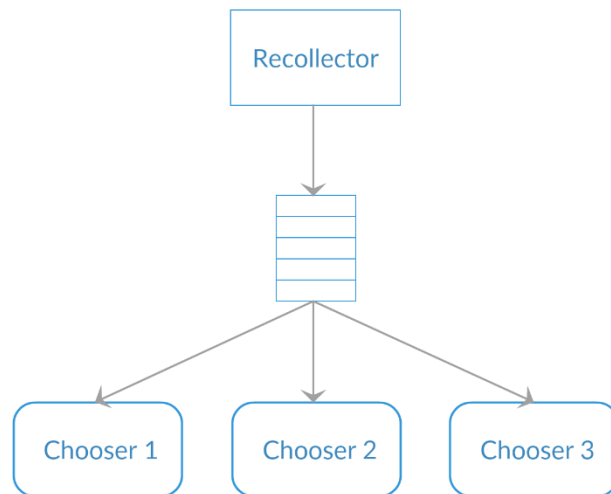
Patrones de canales

A continuación, se exponen los patrones de canales de mensajería que han sido usados o tenidos en cuenta a la hora de realizar la implementación del sistema.

Canal punto a punto

Para poder leer y procesar completamente todos los mensajes que provienen de la API de *streaming* de Twitter, se ha tenido que hacer uso de tres *choosers* simultáneos en la aplicación. Se ha habilitado un canal donde los *tweets* se van almacenando a la espera de ser leídos.

El canal permite que haya varios *choosers* consumiendo mensajes concurrentemente, es decir, son consumidores competidores. Se garantiza que cada *tweet* será consumido solo por un *chooser*.

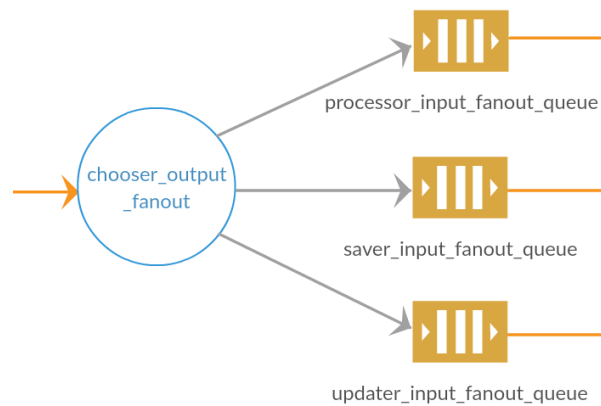


Publicación-suscripción

El primero se utiliza en los WebSockets entre el servidor y los clientes. En este caso cada cliente se suscribe a un tema y el servidor publica mensajes sobre ese tema. En este caso se realiza con los tweets que se deben mostrar en tiempo real en la interfaz.

Existe otra aplicación de este patrón en la aplicación. Esta vez está implementada con *RabbitMQ*

gracias a unas colas en modo *fanout*. Los *chooser* dejan pasar los tweets que nos interesan. Estos tweets se triplican para el *Saver*, el *Updatery* y el *Processor 1*



Canal con tipo de datos

Este patrón resuelve el problema de que el emisor pueda enviar datos que el suscriptor sepa como procesar. En este caso sabemos que todos los datos que van a viajar por el canal son POJOs de JAVA serializados. Más concretamente se trata de Tweets.

Adaptador de canal

En este caso queríamos conectar una aplicación al sistema de mensajería. En este caso es la aplicación de *streaming* de Twitter. Se usa un adaptador de canal que accede a la API de *streaming* y publica los tweets como objetos de Java serializados.

Patrones de mensajes

Mensaje evento

Los módulos de la aplicación *Saver*, *Updater* y los dos *Processor* son elementos que reaccionan cuando llegan mensajes desde el bróker de AMQP. Por lo tanto, podemos considerar esta arquitectura como un patrón de mensajes de evento.

Patrones de enrutado de mensajes

Filtro de mensajes

Este patrón se utiliza para evitar que un componente reciba mensajes que no le interesan. Esto es el ejemplo claro de lo que hacen los *chooser*. Se usa un filtro de mensajes, que elimine los tweets no deseados del canal a partir de ciertos criterios. En este caso, los criterios son modificables en tiempo de ejecución.

Patrones de transformación de mensajes

Enriquecedor de contenido

Se hace uso de este patrón para enriquecer el Tweet original que viene de Twitter. Se cambia la estructura del objeto para permitir un acceso más cómodo a los datos que contiene. Esto se hace por ejemplo en el texto original, se modifica la estructura del objeto para permitir modificar este texto.

Filtro de contenido

Otra manera de ver el trabajo de los *choosers*. Realmente se hace un filtro por el contenido del texto de los tweets. Esto hace que este patrón esté reflejado en el sistema.

Modelo arquitectural

Patrón repositorio

El patrón repositorio se utiliza de manera constante en esta aplicación. Esto es debido a que la mayoría de componentes hacen consultas a una base de datos. Estas consultas se realizan a través de los llamados repositorios. Estos repositorios abstraen las consultas a la base de datos. A través de estos repositorios se lee la información relativa a Tweets, Usuarios, Conexiones, Participantes...

Estos repositorios son una abstracción proporcionada por la integración de Spring con MongoDB. Se expone una interfaz de operaciones varias sobre los elementos de cada repositorio.

La base de datos es externa, es decir, está en una máquina ajena al sistema. En este caso concreto, se trata de una base de datos MongoDB alojada en una máquina virtual de Microsoft Azure.

Puntos cumplidos

A continuación, se presentan los puntos que se han cumplido en la realización de este proyecto.

Aprobado

- Se ajusta a la arquitectura de referencia propuesta.
- Tiene al menos un Tweet Processor
- Tiene al menos un Messaging Broker AMQP (p.ej. RabbitMQ)
- Expone la funcionalidad del componente Dashboard mediante una Web API que no viola los principios de la arquitectura de la Web
- Soporta correctamente a varios usuarios simultáneos
- Se despliega y ejecuta en al menos tres máquinas virtuales cloud diferentes
- Incluye al menos tres vistas arquitecturales, una de distribución en estilo de despliegue/instalación, otra de módulos y otra de conectores, y estas son una descripción fiel del sistema

Notable

- Cumple los mínimos exigidos para el aprobado.
- Tiene al menos dos Tweet Processors.
- Permite que cada Tweet Processor sea configurado en tiempo de ejecución mediante una API pública.
- Requiere una autenticación de tipo OAuth o similar para poder acceder a la API de configuración de los Tweet Processors
- Expone la funcionalidad del componente Dashboard mediante una API RESTful (hasta donde tenga sentido)
- Se despliega y ejecuta en al menos cinco máquinas virtuales cloud diferentes

- Se señalan todos los patrones EIP que se usan en el sistema
- Se realiza un plan de explotación económico del sistema para un año de costes operativos (analizar varios proveedores públicos)
- La API se documenta utilizando alguna especificación industrial actual (p.e. <https://github.com/OAI/OpenAPI-Specification>) o el esquema de algún proveedor de API reconocido (p.ej. Google)

Sobresaliente

- Cumple los mínimos exigidos para el notable
- Permite que el componente Tweet Chooser sea configurado en tiempo de ejecución mediante una API pública
- Requiere una autenticación de tipo OAuth o similar para poder acceder a la API de configuración del Tweet Chooser
- Se despliega y ejecuta en al menos cinco máquinas virtuales cloud diferentes situadas en al menos dos proveedores de cloud diferentes.
- Es robusto y/o escalable

Plan de explotación

Se va a dividir el despliegue de la aplicación en tres principales componentes. Cada uno de ellos está alojado en los servidores de proveedores diferentes. Para el presente estudio del precio de explotación se contemplan opciones que nos permitan ser robustos ante posibles fallos de las máquinas.

Los tres bloques en los que se va a desplegar la aplicación son:

- Máquinas para los microservicios de *Spring*
- Base de datos
- *RabbitMQ*

Máquinas virtuales

Se van a desplegar siete máquinas virtuales para alojar a cada uno de los servicios que componen el sistema global.

Para ello vamos a necesitar unas máquinas con procesamiento de cálculo y algo de memoria. No es necesario almacenamiento local debido a que todos los datos están alojados fuera de la máquina.

Para ello, hemos considerados tres proveedores diferentes:





- *Google Cloud*
- *Amazon Web Services*
- *Microsoft Azure*

Google Cloud

Se piden siete máquinas *n1-standard-4*. Estas máquinas tienen cuatro CPUS y 15 GB de memoria RAM. Como región de despliegue se elige Europa debido a que la mayoría de clientes van a ser de esta región.

También se incluye en el precio un balanceador de carga con una estimación de paso de datos de 10GB al mes.

A continuación, se puede ver una imagen con más detalles.

Compute Engine		
7 x		
5,110 total hours per month		
VM class: regular		
Instance type: n1-standard-4		
Region: Europe		
Commitment term: 1 Year		
Estimated Component Cost: \$8,073.88 per 1 year		
Load Balancing (global)		
Europe		
Forwarding rules: 20		
Network ingress: 10 GB		
\$1,533.96		
Total Estimated Cost: \$9,607.84 per 1 year		

A este precio habría que sumarle un 10% extra para las máquinas virtuales. Así tendríamos en cuentas posibles caídas.

Con todo esto, el precio de alojar nuestra aplicación en *Google Cloud* sería de:

$$8073.88 + 8073.88 * 0.1 + 1533.96 = 10415.23\text{€}$$

[Amazon Web Services](#)

Se piden siete máquinas *m4.xlarge*. Estas máquinas tienen cuatro CPUS y 16 GB de memoria RAM. Como región de despliegue se elige Europa (Irlanda) debido a que la mayoría de clientes van a ser de esta región.

También se incluye en el precio un balanceador de carga con una estimación de paso de datos de 10GB al mes.

A continuación, se puede ver una imagen con más detalles.

[-]	Amazon EC2 Service (US-East)		\$ 15166.38
	Compute:	\$ 0.00	
	Reserved Instances (One-time Fee):	\$ 15148.00	
	Elastic LBs:	\$ 18.30	
	Data Processed by Elastic LBs:	\$ 0.08	
[+]	Amazon EC2 Service (Europe)		\$ 2664.58
[-]	AWS Support (Business)		\$ 1546.85
	AWS Support Plan Minimum:	\$ 100.00	
	Support for Reserved Instances (One-time Fee):	\$ 1446.85	
	Free Tier Discount:	\$ -18.93	
	Total One-Time Payment:	\$ 19238.85	
	Total Monthly Payment:	\$ 120.03	

Con todo esto, el precio de alojar nuestra aplicación en *Amazon Web Services* sería de:

$$19238.85 + 120.03 * 12 = 20679.21\text{€}$$

Microsoft Azure

Se piden siete máquinas D3 v2. Estas máquinas tienen cuatro CPUs y 14 GB de memoria RAM. Como región de despliegue se elige Europa Occidental debido a que la mayoría de clientes van a ser de esta región.

A continuación, se puede ver una imagen con más detalles.

Service type	Custom name	Region	Description	Estimated Cost
Virtual Machines	Virtual Machines	West Europe	7 Estándar máquinas virtuales, tamaño Promoción de D3 v2 (4 núcleos, 14 GB de RAM, 200 GB en disco): 744 horas	€1.054,06
Support			Support	€0,00
			Monthly Total	€1.054,06
			Annual Total	€12.648,69
Disclaimer				
All prices shown are in Euro (€). This is a summary estimate, not a quote. For up to date pricing information please visit https://azure.microsoft.com/pricing/calculator/ This estimate was created at 5/20/2017 8:27:56 PM UTC.				

A este precio habría que sumarle un 10% extra para las máquinas virtuales. Así tendríamos en cuentas posibles caídas.

Con todo esto, el precio de alojar nuestra aplicación en *Microsoft Azure* sería de:

$$12648.69 + 12648.69 * 0.1 = 13913.56\text{€}$$

RabbitMQ

Para desplegar el bróker de mensajes *AMQP* se ha elegido el servicio *CloudAMQP*. En esta plataforma existen diferentes niveles de servicio en función de las necesidades. La mejor opción para nuestra aplicación es la *Big Bunny* debido a que es la más barata sin restricciones de número de colas ni mensajes.

Ofrece 1000 conexiones simultáneas y 1000 mensajes por segundo. Tiene un coste de 99 \$/mes, esto es 1188\$ al año.

MongoDB

Para desplegar la base de datos *MongoDB* se ha elegido el servicio *mLab*. En esta plataforma existen diferentes niveles de servicio en función de las necesidades.

Se calcula que se va a necesitar unos 60GB durante este año para almacenar la información generada por la aplicación. Para ello vamos a optar por un servicio de 4GB de RAM y 60GB de almacenamiento.

Tiene un coste de 360 \$/mes, esto es 4320\$ al año.

Total

		Google Cloud	Amazon Web Services	Microsoft Azure		
	Máquinas	\$10.415,23	\$20.679,21	\$13.913,56		
	Broker AMQP	\$1.188,00	\$1.188,00	\$1.188,00		
	MongoDB	\$4.320,00	\$4.320,00	\$4.320,00		
	Total	\$15.923,23	\$26.187,21	\$19.421,56		

A la luz de los resultados, se va a optar por usar los servicios de Google Cloud a un precio anual de 15923,23\$.

Diagramas

Diagrama de despliegue

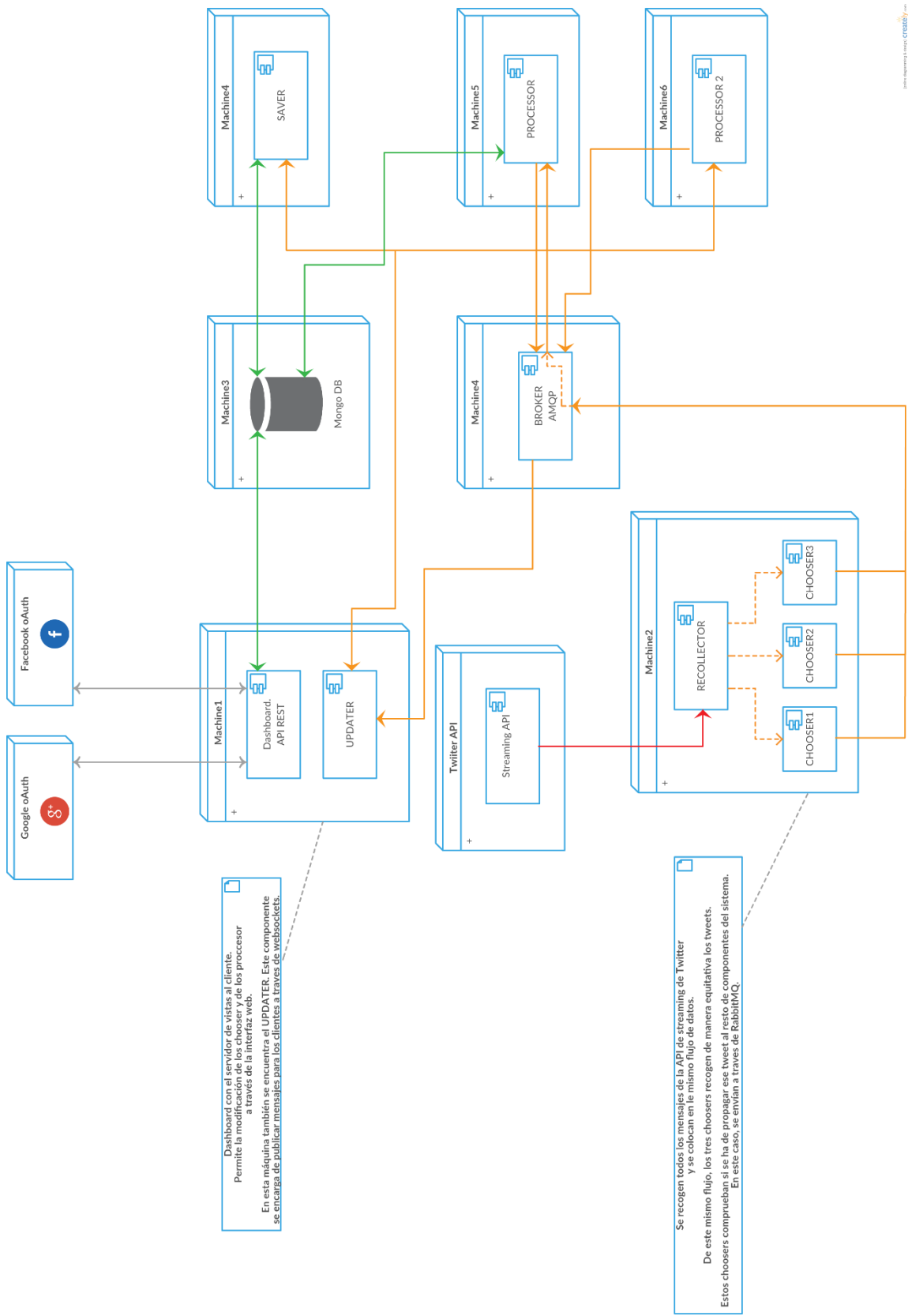


Diagrama de conectores

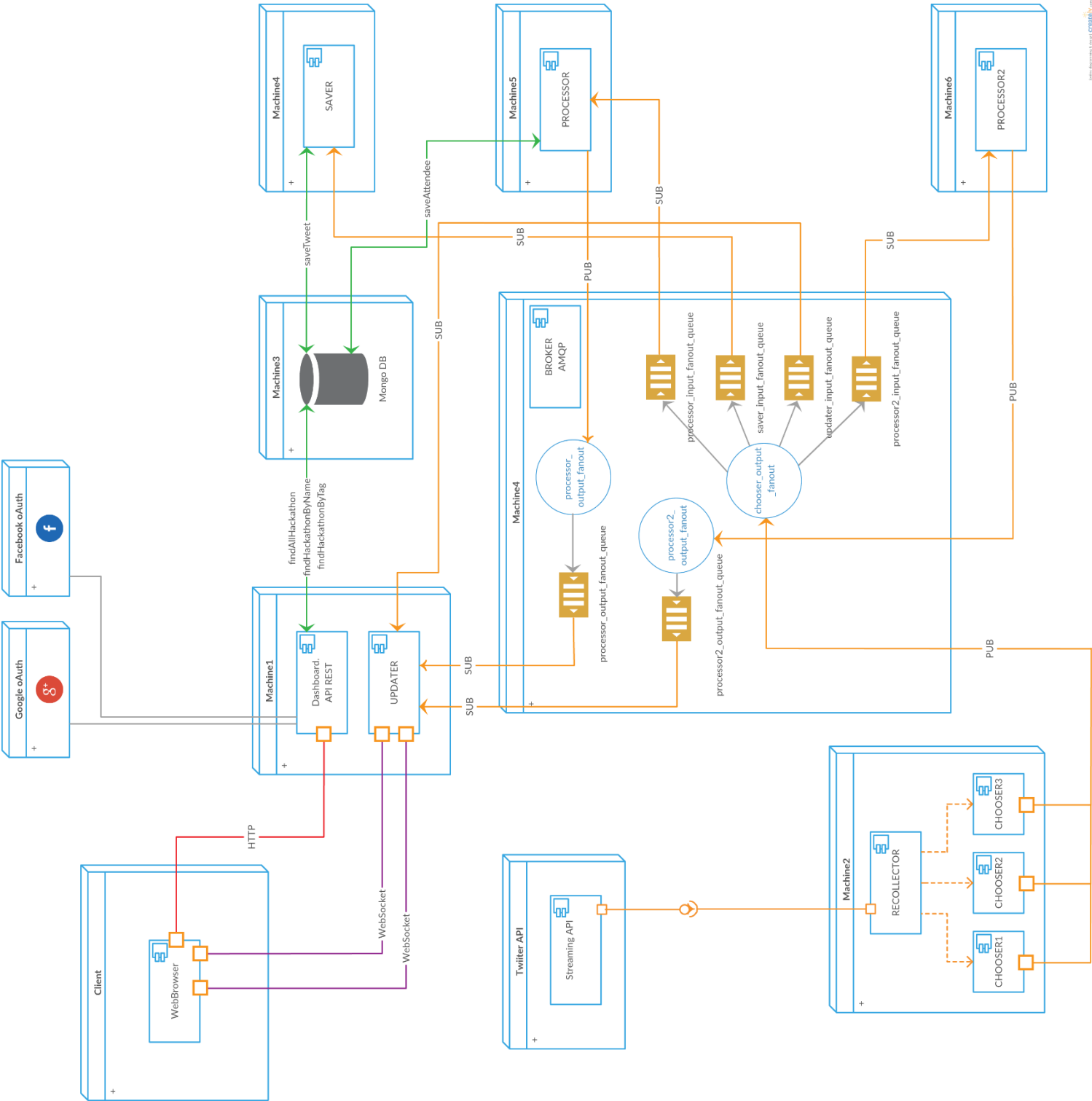
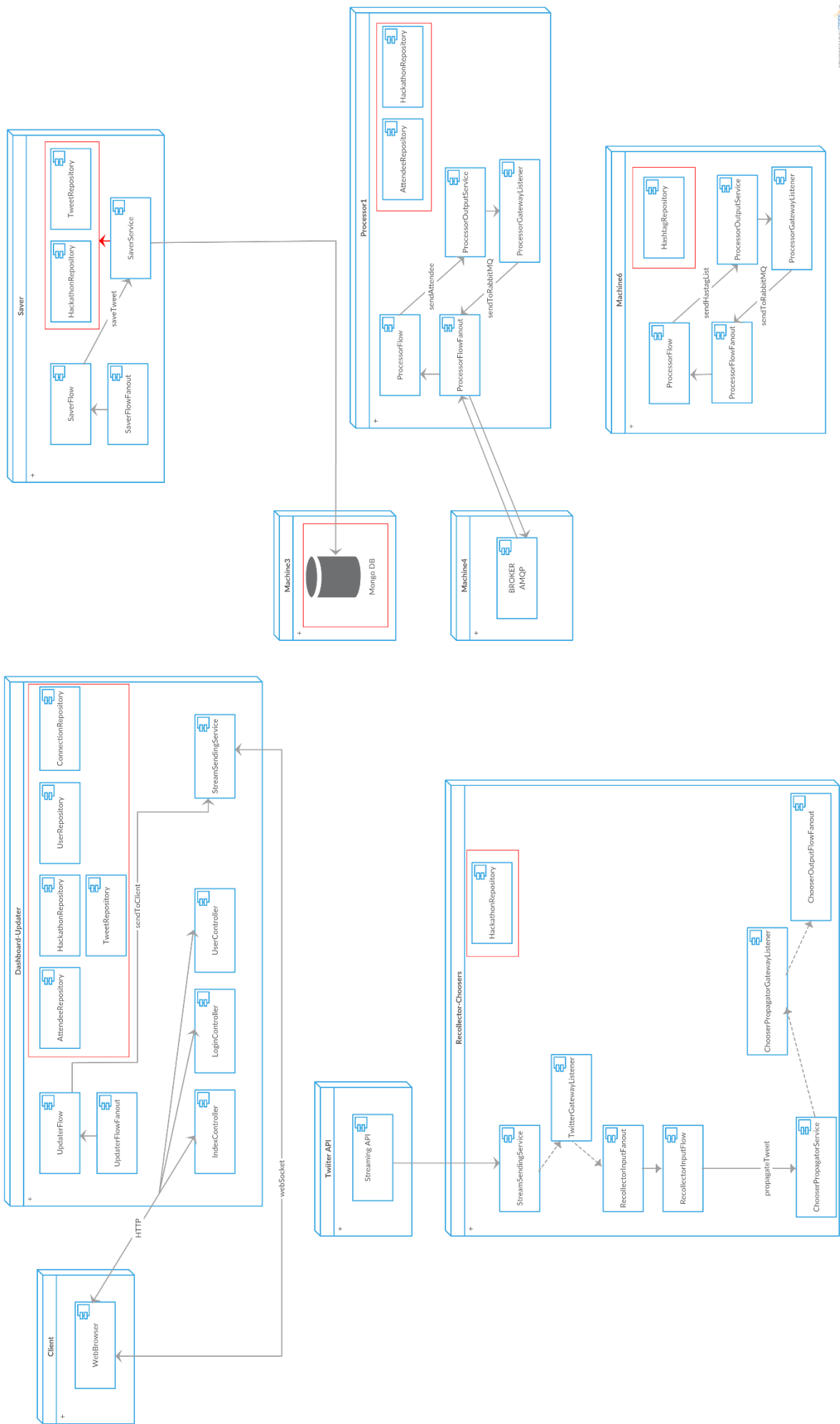


Diagrama de componentes/módulos



Github

El código fuente se puede ver en la siguiente URL:

<https://github.com/jorcox/TweetRegistration>