

Diseñando Sistemas Altamente Escalables

¿Que aprenderemos?

- Principios core de micro servicios
- Construyendo los bloques de una arquitectura de micro servicios
- Administración de la data de micro servicios
- Anti-patrones que debemos evitar
- Factores de éxito
- Desplegando micro servicios
- Planes para convertir monolitos a micro servicios

¿Que son los micro servicios?

Los micro servicios son aplicaciones o servicios pequeñas, pobremente acopladas que pueden fallar de forma independiente sin afectar al resto.

Si un micro servicio falla, solo una simple función o proceso en el sistema estará indisponible, mientras el resto del sistema continúa trabajando...

Principios de micro servicios

- Cada micro servicios debería ser solo responsable de una simple función o proceso.
- Los micro servicios no deberían compartir código o data.
- La independencia y autonomía son más importantes que la reusabilidad de código.
- Los micro servicios no deberían tener permitido comunicarse directamente entre ellos. Ellos deberían hacer uso de un bus de eventos/mensajes para comunicarse entre ellos.

Beneficios de micro servicios

- Promueve la modularidad, haciendo mas sencillo el entendimiento, desarrollo y pruebas del sistema.
- Reduce la complejidad teniendo pequeños código base por micro servicio.
- Te permite actualizar funcionalidad con nada o mínimo efecto en el resto del sistema.
- Reduce enormemente la chance de romper algo en una parte no relacionada del sistema.
- Permite una mayor colaboración en el equipo ya que esta trabajando en el mismo sistema al mismo tiempo.

Beneficios de micro servicios

- Habilita la entrega continua y desarrollo de grandes y complejas aplicaciones aplicando el principio de “Divide y vencerás”.
- Los servicios pueden ser desplegados de forma independiente sin tener que esperar por que el sistema completo sea publicado.
- Este crea una arquitectura que es altamente escalable.
- Permite el deployment a multiples cloud y/o entornos de infraestructura on-premise
- Toma ventaja de tecnologías emergentes (frameworks, lenguajes de programación, etc.) mientras evoluciona el sistema existente.
- Permite a nuevos miembros del team ser mas productivos desde que inician a desarrollar nueva funcionalidad sin tener que aprender el sistema completo.

Anti-patrones

- Todo debería ser micro excepto la base de datos
- Los micro servicios mágicamente solucionan pobres prácticas de desarrollo
- No hay necesidad de coordinación entre equipos de desarrollo
- Tener el foco en las tecnologías detrás de los micro servicios

1. Todo debería ser micro excepto la base de datos

- Si tu tienes una simple base de datos, esta ser convertirá en un simple punto de falla.
- Los micro servicios no pueden fallar independientemente si ellos están todos acoplados a la misma simple base de datos.

2. Los micro servicios mágicamente solucionan pobres practicas de desarrollo

- Si malas practicas de desarrollo o malos hábitos de despliegue son transferidos de una vieja arquitectura a una arquitectura de micro servicios podría causar mas problemas que nada.
- Buenas practicas de codificación y despliegue deben ser siempre una prioridad en lugar de ser una elección de arquitectura.

3. No hay necesidad de coordinación entre equipos de desarrollo

- Desde que los micro servicios son un nuevo concepto para muchos equipos, debería haber siempre coordinación entre equipos.
- Es buenísimos que un arquitecto de software se asegure de que los micro servicios sean diseñados acorde a los mismos altos estándares y buenas prácticas que existen en todos los equipos.
- Debería haber una forma estándar y uniforme en como son los micro servicios diseñados, estándares de codificación que deben seguirse y guías y estándares de como los micro servicios deberían ser documentados y monitoreados.

4. Poner el foco en las tecnologías detrás de los micro servicios

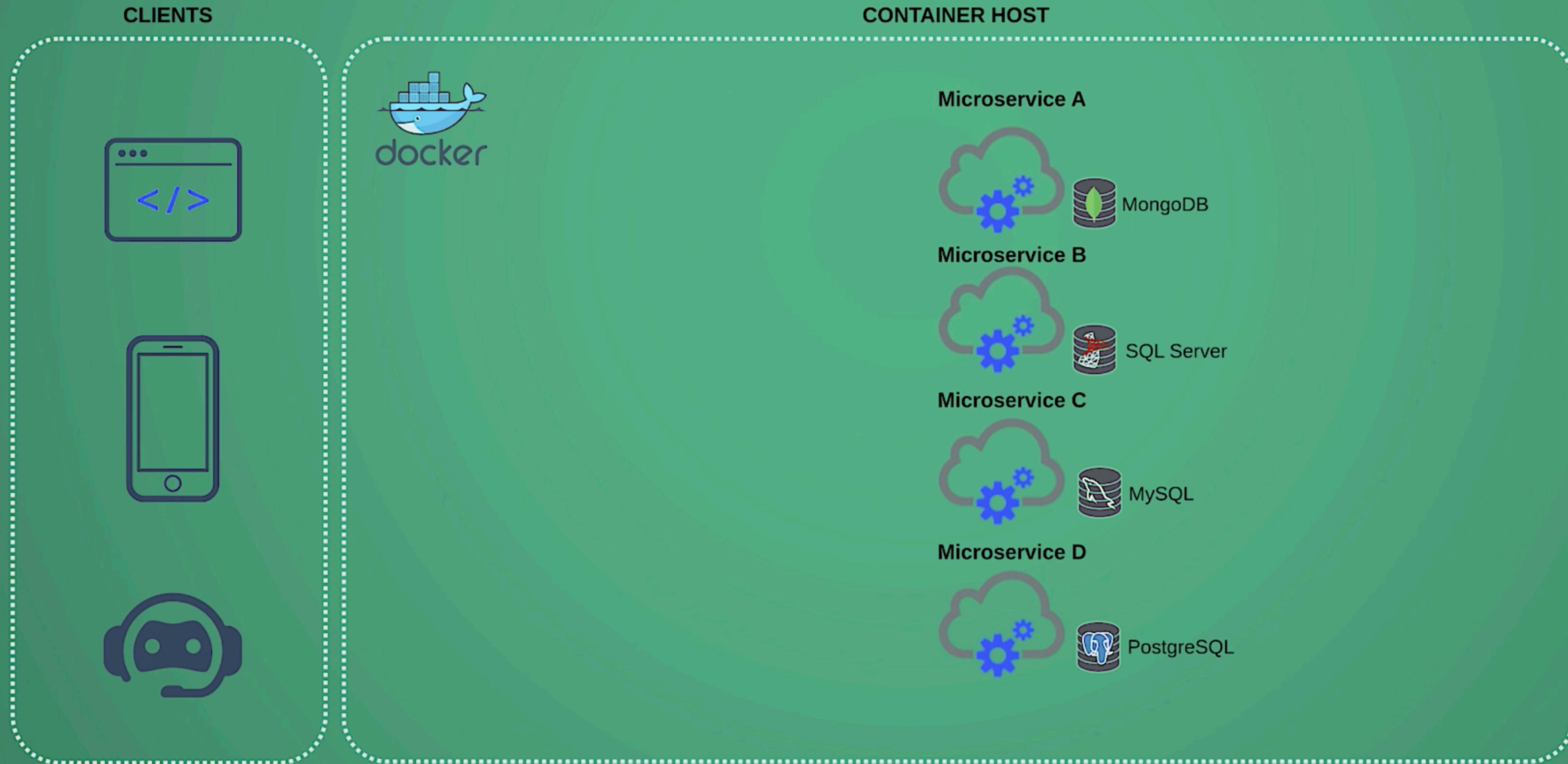
- Los micro servicios nos dan la flexibilidad en la elección de herramientas y tecnologías.
- Hay magnificas herramientas y tecnologías disponibles para desarrollar y administrar micro servicios, pero, ellos deberían nunca ser tu principal foco.
- El foco debería ser en como la funcionalidad del sistema debería ser descompuesto en micro servicios y como definir el propósito de cada micro servicio.
- Tu no deberías atarte a una particular herramienta o tecnología.

Bloques de construcción

Bloques de construcción

- Vista del despliegue de la arquitectura de micro servicios
- Micro servicios como RESTful APIs
- Comunicación cliente-a-micro servicios, a través, de un API Gateway
- Comunicación orientada a eventos entre micro servicios usando un bus de eventos.
- Asegurando micro servicios

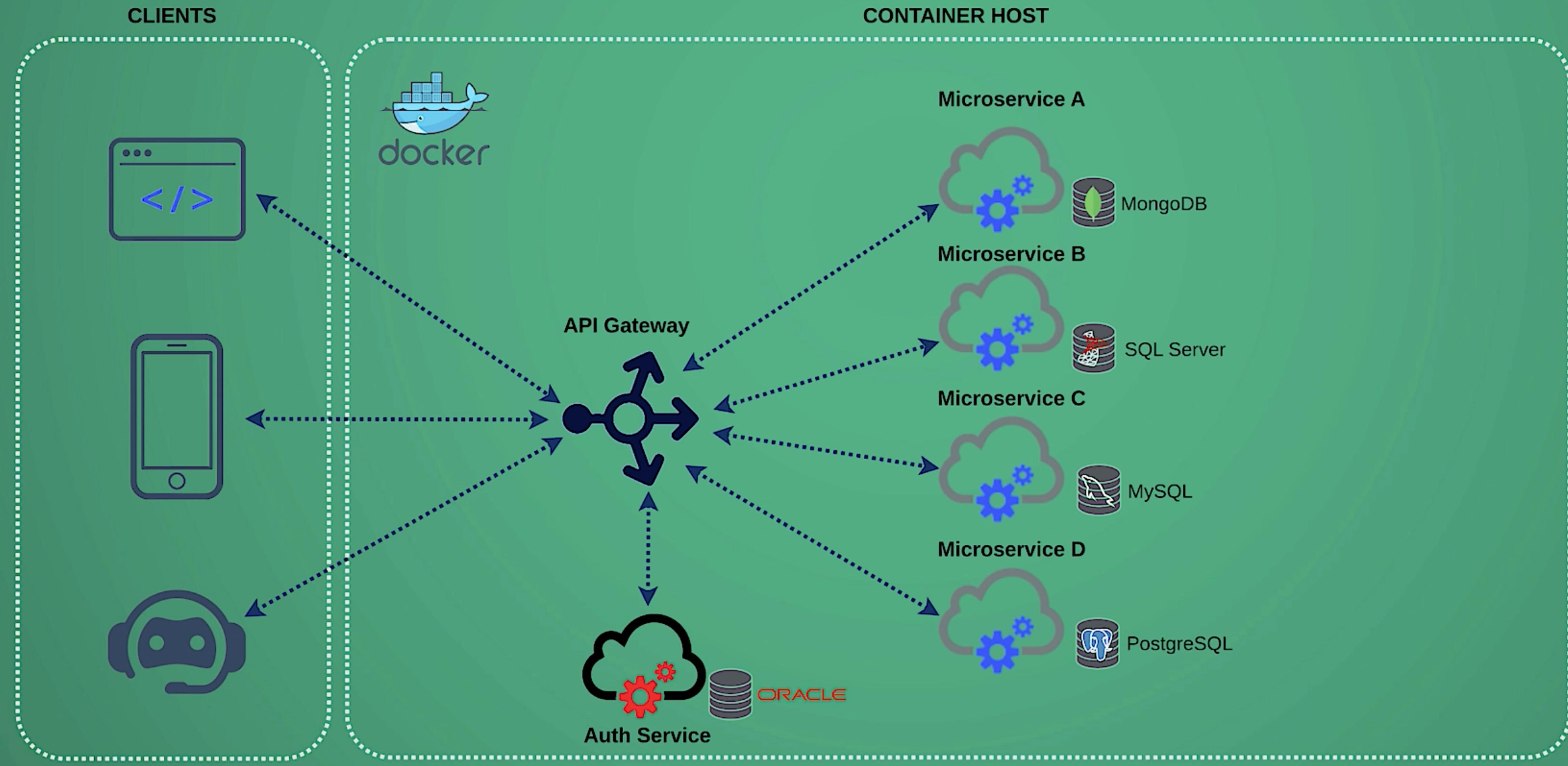
Typical Microservices Architecture



Típica arquitectura de micro servicios

- Un API gateway permite comunicación entre clientes y micro servicios sobre HTTP
- En lugar de llamar a cada micro servicio directamente, los clientes deben hacer sus requests a uno o mas API gateways que ruteará los HTTP requests al micro servicio deseado, que ejecutará el comando o query request.

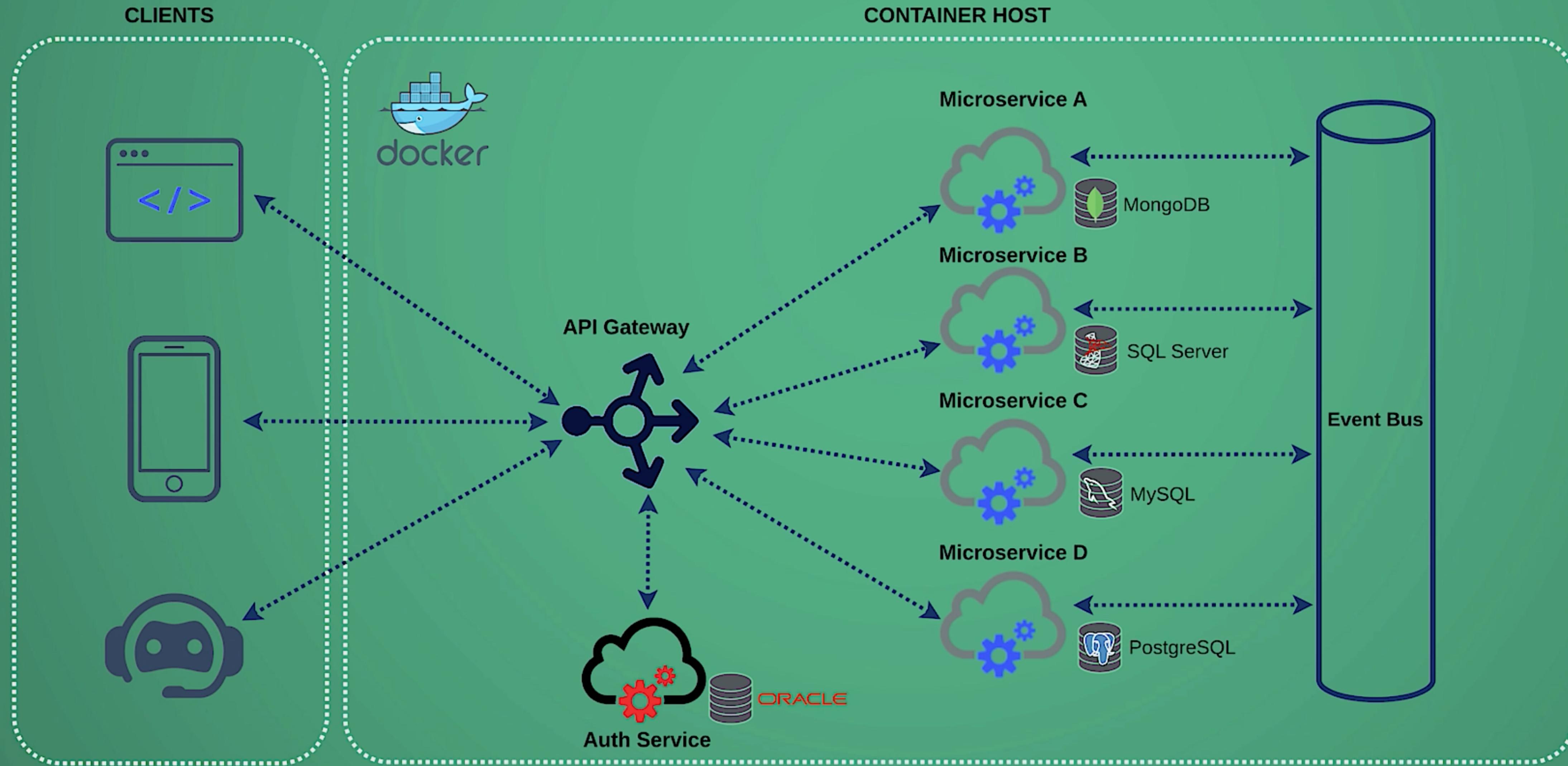
Typical Microservices Architecture



Servidor de Autorización

- El access token necesita ser enviado en cada HTTP request para lograr acceso seguro a cada micro servicio
- Una aplicación cliente debería hacer dos requests HTTP. El primero para obtener el access token. Y el segundo para hacer el request a el deseado micro servicio vía un API Gateway.
- Los access tokens son generalmente activos por un tiempo configurable. Sin embargo, si por ejemplo, un token es valido por 5 minutos, un cliente puede hacer multiples requests antes de que el token expire, eliminando la necesidad de hacer un nuevo request por token en cada llamada.
- Debe haber un balance entre la frecuencia de accesos y tiempo de expiración, siempre tener en mente el tiempo de expiración muy corto, para hacer más seguro el request.

Typical Microservices Architecture



¿Qué es un RESTful API?

- Un RESTful API es una API web o servicio que esta basado en el estilo de arquitectura conocido como Representational State Transfer.
- REST define como las aplicaciones clientes pueden comunicarse con un RESTful API sobre HTTP.
- El request de un cliente consiste de un URI, un verbo HTTP, un request header, y un body opcional.

¿Qué es un RESTful API?

- Los verbos HTTP más usados son...
 - POST
 - PUT
 - PATCH
 - DELETE
 - CRUD

Verbos HTTP

- POST
 - Crear un nuevo recurso
- GET
 - Recuperar un recurso específico
- PUT
 - Actualizar un recurso específico
- PATCH
 - Hacer un parcial update al recurso
- DELETE
 - Eliminar un recurso específico

GET vs POST

- POST es más seguro que el GET
- GET es menos seguro porque el request de datos es enviado como parte del URI, lo cual lo hace visible para todos, y es guardado en la historia del navegador
- Con un POST, la data del request es enviado como parte del body del request, y no es incluido en el request URI. Esto es mas seguro desde que el request de datos no es guardado en el navegador y tampoco es cacheado.

GET vs POST

- Otras consideraciones para usar POST en lugar de GET
- No puedes usar un tipo de objeto completo como parámetro en un método GET
- El método GET solo permite caracteres ASCII, mientras que los métodos POST no tienen restricciones y soportan data binaria.
- La maxima longitud de un request URL es 2048 caracteres. Los métodos POST no envían la data del request en el URL, en verdad no hay restricción en la longitud de la data.

¿Porque RESTFUL APIs?

- Simplicidad - Es facil de entender, desde que los verbos HTTP son basados en CRUD
- REST es diseñado para ser stateless y separar los concerns del cliente y el servidor. Esto significa que el servidor no tiene que saber el estado del cliente y el cliente no tiene que preocuparse del estado del servidor.
- Las lecturas REST pueden ser cacheadas para mejor performance y escalabilidad.
- REST soporta muchos formatos de datos, pero, el predominante uso de JSON permite mejor soporte para clientes browser.
- Muchas compañías como Google, Amazon y Microsoft proveen sus APIS en la forma de RESTful APIs.

¿Qué es JSON?

- JSON es una abreviación de JavaScript Object Notation
- JSON es un texto o formato de datos ligero que es sencillo para leer por humanos y computadoras. Es generalmente usado para intercambio de datos entre aplicaciones y web APIs o servicios.
- Aunque su origen se dió en lenguaje de programación JavaScript, es independiente del lenguaje.

Http Status Codes

- 1xx - informativo
- 2xx - éxito
- 3xx - Redirección
- 4xx - Errores de Cliente
- 5xx - Errores de servidor

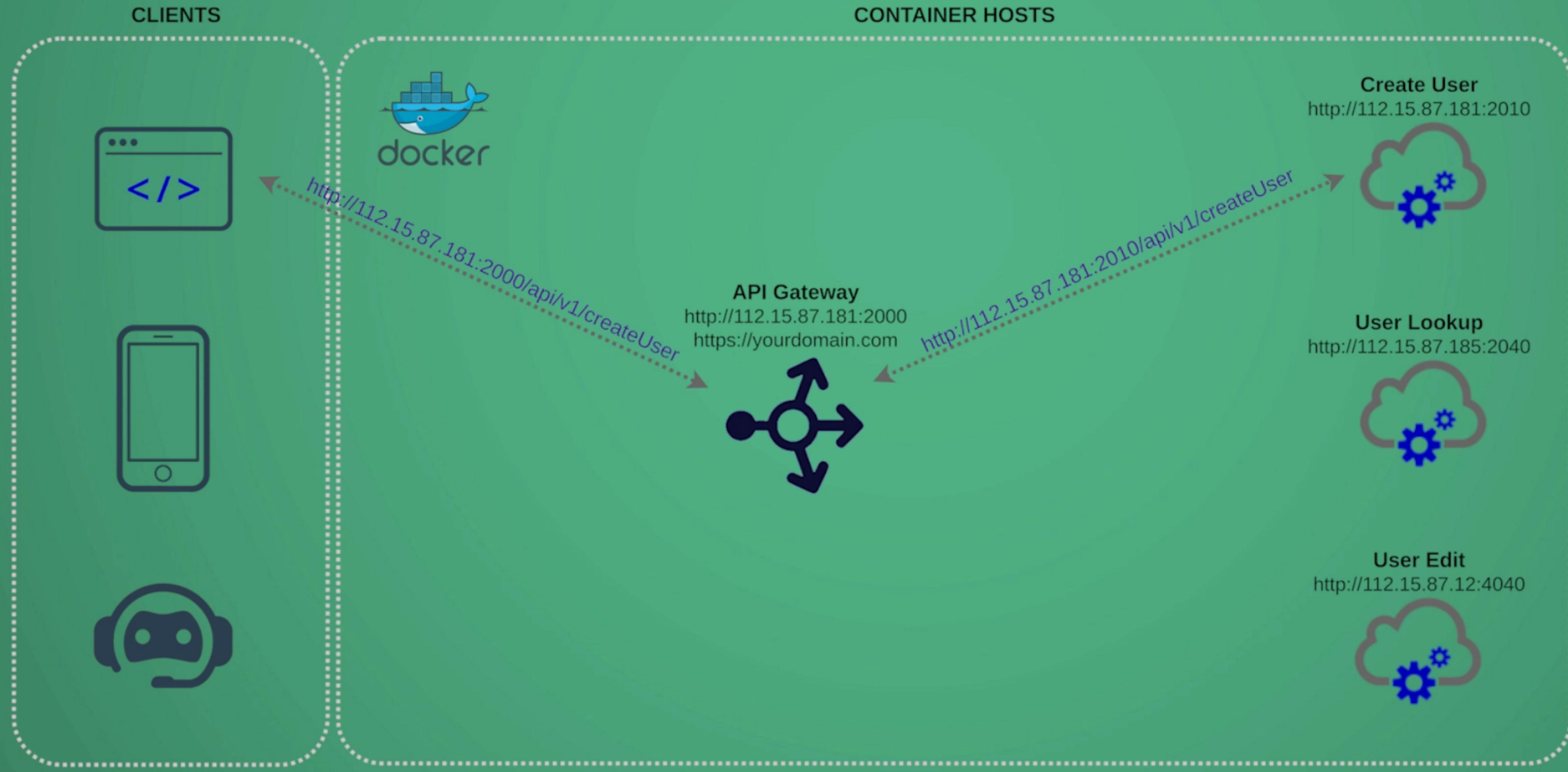
Http Status Codes

- 200 - el request fue procesado sin problemas
- 201 - recurso creado satisfactoriamente
- 204 - request retorno sin contenido
- 400 - bad request
- 401 - no autorizado
- 403 - forbidden (prohibido)
- 405 - método no permitido
- 500 - internal server error

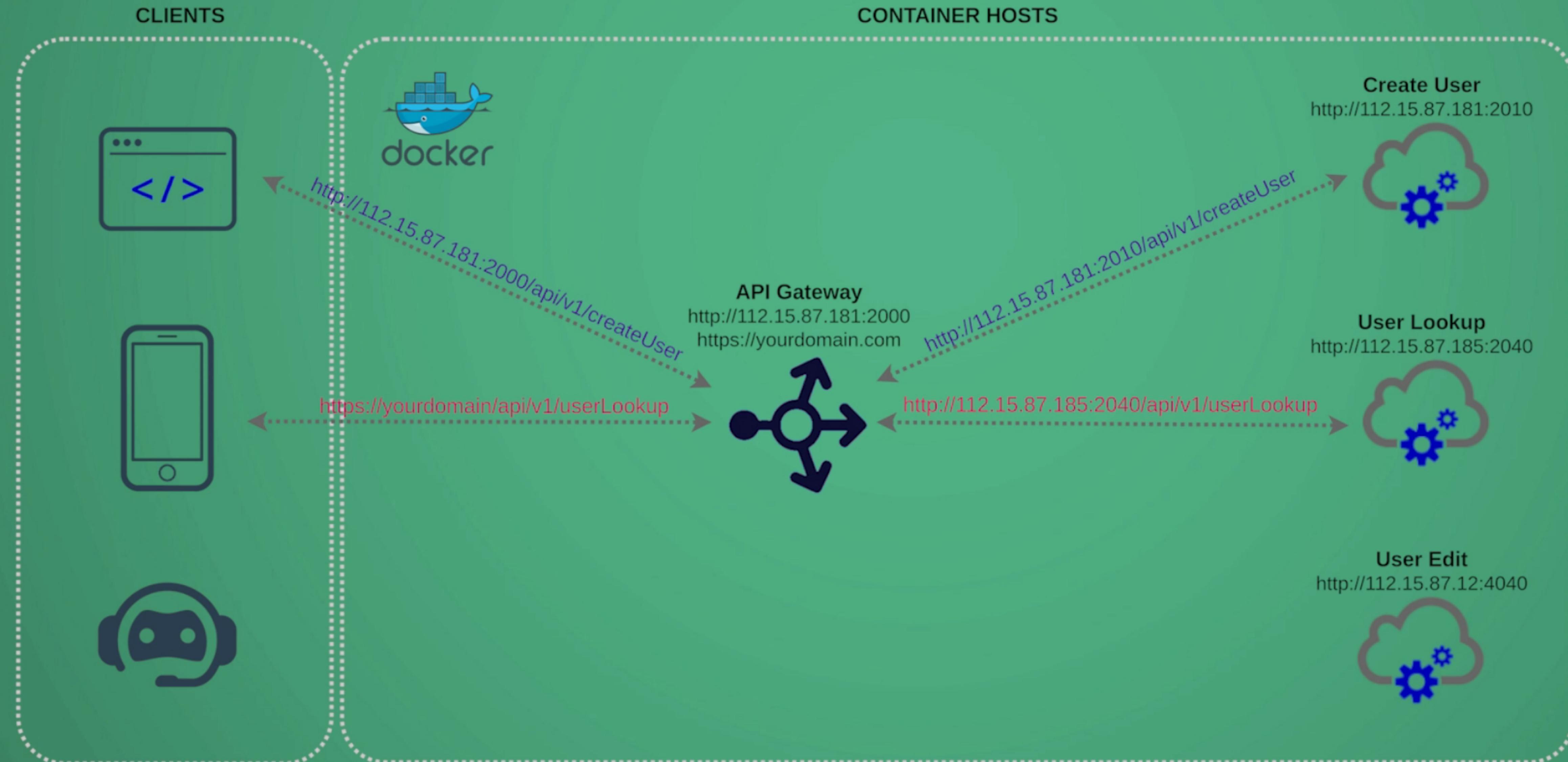
¿Qué es un API Gateway?

- Un API Gateway crea un punto de entrada único que las aplicaciones clientes pueden usar para acceder a los micro servicios.
- Esto actúa como un reverse proxy que rutea los requests HTTP que son hecho por clientes a los microservicios backend deseados.
- API Gateways pueden también ejecutar otras importantes funciones como autenticación del cliente, balanceo de carga y terminación SSL
- El API Gateway pueden también limitar el tipo de verbos HTTP que son permitidos por un específico gateway route.

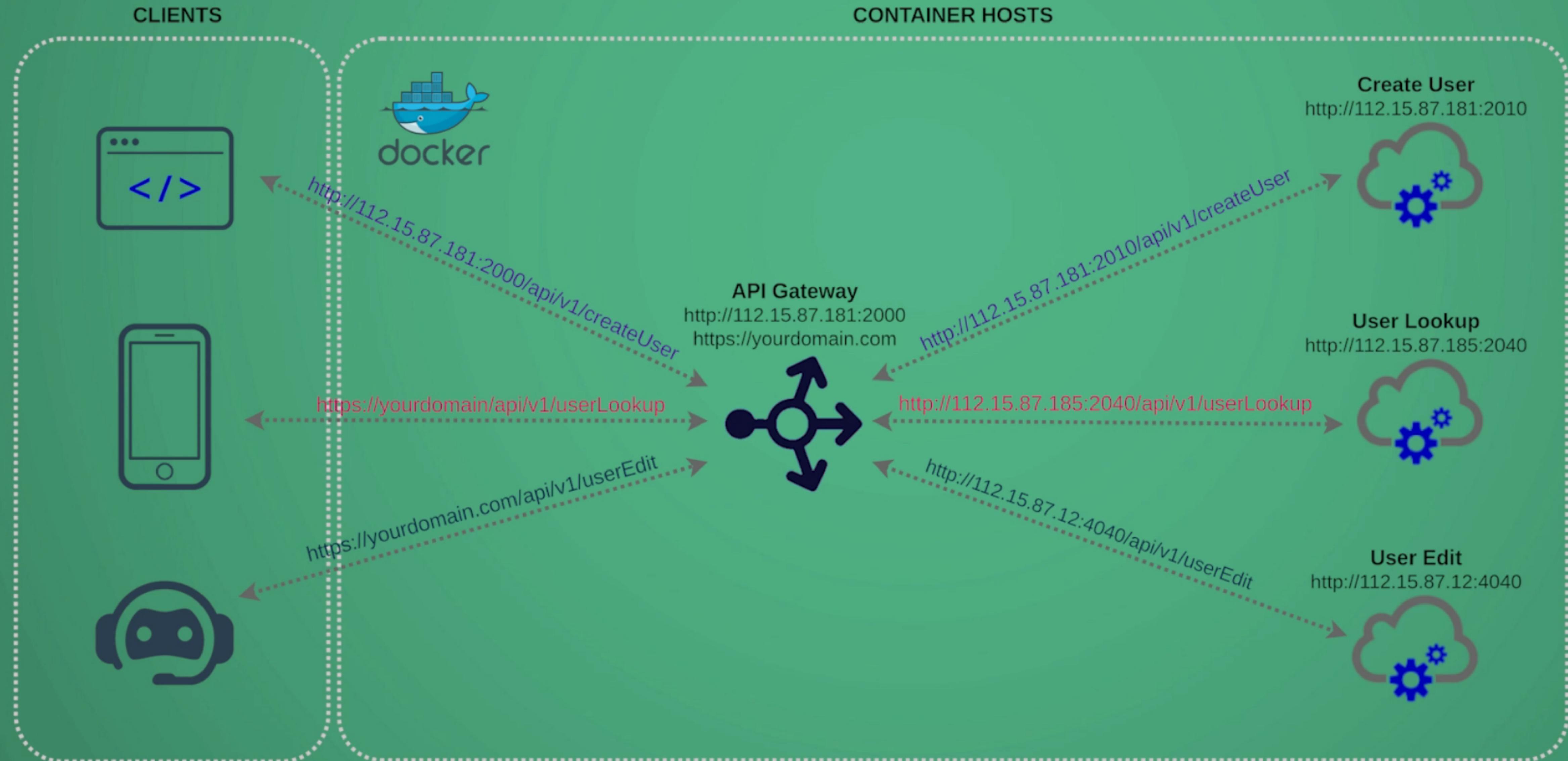
API Gateway Routing Example



API Gateway Routing Example



API Gateway Routing Example



Problemas con acceso directo al cliente

- Incrementa la complejidad de integración con el cliente si es que el cliente tiene que mantener seguimiento de las diversos endpoints de micro servicios.
- Considerar que los micro servicios pueden ser migrado a una nueva versión que reemplaza el servicio al cual un cliente puede llamar.

¿Qué es un API Gateway?

- Si el servicio es migrado, o una nueva versión es liberada, nosotros podemos simplemente hacer que el API Gateway apunte al nuevo endpoint o versión.
- De esta manera, la aplicación cliente puede aun hacer el mismo request sin tener que saber si el servicio fue movido o actualizado.
- Si el schema JSON request y response sigue siendo el mismo, el cliente no se entera si ha cambiado el backend.

Problemas con acceso directo al cliente

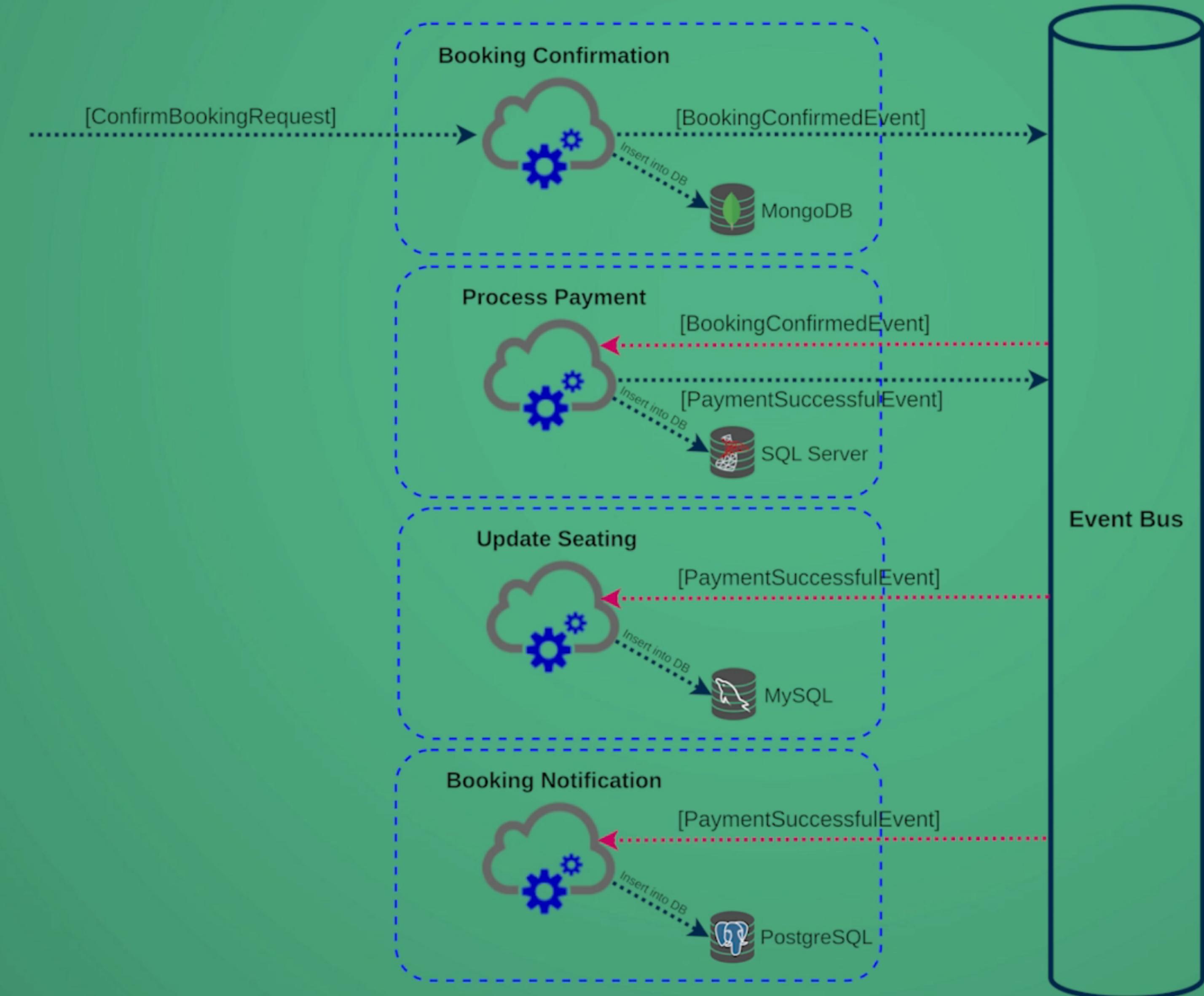
- Los clientes tienen que implementar su propio load balancing y detección de fallas.
- Si clientes desean acceder a múltiples servicios publicados, entonces todos esos servicios pueden tener sus propios constraints de seguridad, incluyendo terminación SSL y autenticación.
- Exponer servicios de forma pública, los expone al riesgo de ser atacados.
- Se puede implementar restricciones en API Gateways, lo cual no es posible con acceso directo al cliente.

Comunicación orientada a eventos
usando un bus de eventos

¿Qué es un bus de eventos?

- Este permite a los micro servicios comunicarse unos con otros sin tener que saber de la existencia de ellos.
- Este tipo de comunicación es basada en el patrón Publish/Subscribe, el cual es similar al patrón Observer. Por lo tanto, con el patrón Observer el publicador u observable broadcasts cambia directamente a suscriptores u observers.
- El Bus de Eventos toma el rol de middleman y se sitúa entre el publicador y suscriptor.
- De esta manera, los microservicios que publican eventos al bus de eventos, no tienen que saber que harán los micro servicios con los eventos publicados, por lo tanto, solo se tiene que asegurar que estén los eventos o mensajes disponibles en el bus de eventos para su posterior proceso.

Event-Driven Communication using an Event Bus



Seguridad en micro servicios

Seguridad en micro servicios

- Usar proveedores de autenticación externos como OAuth 2.0
- Agregar una capa de autenticación en el API Gateway
- Crear tu propio micro servicio de autenticación

1. Usando proveedores de autenticación externos

- OAuth 2.0 es un estándar para autorización en la industria
- OAuth provee clientes con acceso delegado a recursos. Lo cual significa que en lugar de compartir tus credenciales de login, tokens de autorización, estos son generados por un servidor de autorización, y son usados para acceder a los micro servicios.

2. Agregar una capa de autenticación en el API Gateway

- Desde que el API Gateway es el punto de entrada de un micro servicio, muchos argumentan que la autenticación y autorización debería ser responsabilidad del API gateway.
- Mientras otros argumentan que colocar estas responsabilidades en el API Gateway viola los principios core de micro servicios.
- El API Gateway es en si otro micro servicio que debería tener una simple función o proceso. Su propósito core debería ser rutear HTTP requests entre clientes y micro servicios.

3. Crear tu propio micro servicio de autenticación

- Si creas tu propio micro servicio de autenticación, se recomienda para la autenticación de clientes, proveer a ellos un JSON Web Token (JWT) que ellos puedan usar para acceder a sus micro servicios.
- Es recomendable tener un micro servicio separado que toma la responsabilidad de crear usuarios del API.
- Los password del usuario deberían estar encriptados usando un algoritmo de hashing criptográfico, como hashing a password con un salt.
- Un salt es una data random que es usada como entrada al algoritmo hashing, el cual hace casi imposible la desencriptación.
- La función primaria de los salts es defendernos de los ataques.

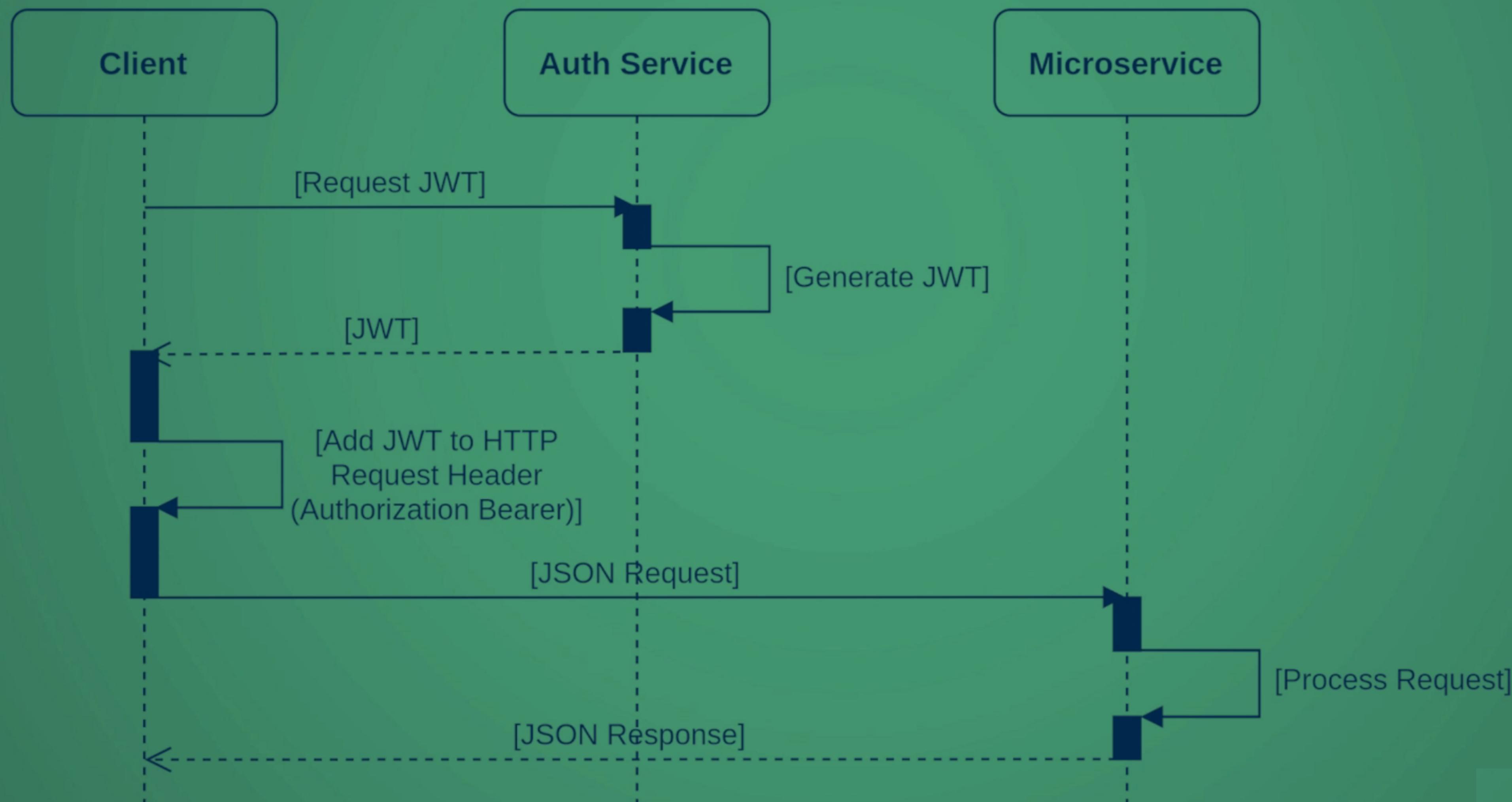
3. Crear tu propio micro servicio de autenticación

- Cuando un usuario requiere un token, en lugar de descripturar su password, el servicio de autenticación debería extraer el salt del password almacenado, y hash el password con el salt extraído.
- Estos nos permite comparar los dos hashed password bit por bit para ver si ellos coinciden, y si es así, se genera el JWT.

JSON Web Token (JWT)

- Un JSON Web Token consiste de un conjunto de claims, los cuales hacen referencia a información en la forma de parejas key/valor (Claim Name/Value) que generalmente usados para autenticación, autorización y para intercambio de información sensitiva.
- JSON Web Tokens son confiables, puesto que ellos son firmados usando un algoritmo HMAC (hash-based message authentication code) o una firma RSA (Rivest-Shamir-Adleman) con SHA-256 (RS256).

Secure Request Procedure

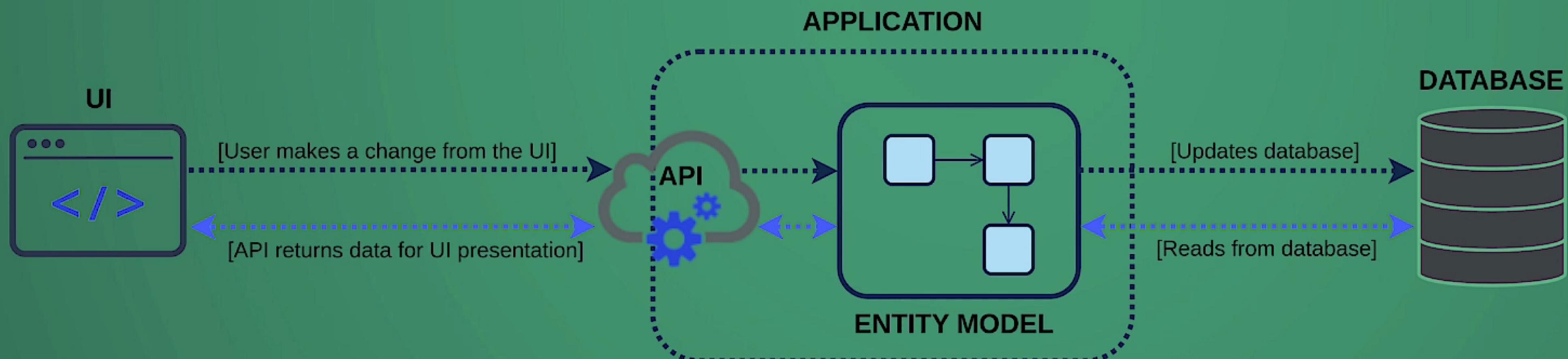


Administración de Datos

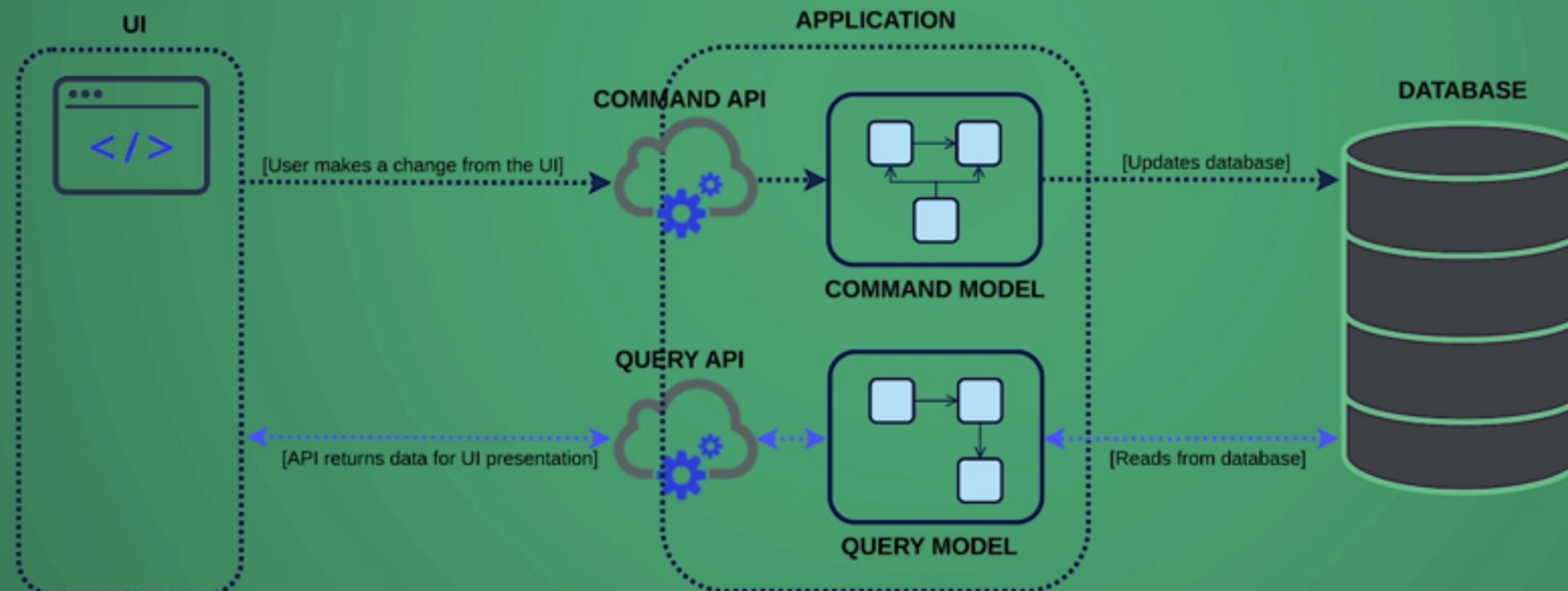
Qué es CQRS?

- CQRS es un patrón de diseño de software que representa un Command Query Responsibility Segregation.
- Commands: Operaciones que modifican el estado de un objeto o entidad.
- Queries: Operaciones que retornan el estado de un objeto o entidad

CRUD in a Traditional Architecture



Basic CQRS: Separate Models for Commands & Queries with Single Database



¿Por que necesitamos CQRS?

- En general, la data es frecuentemente consultada que actualizada.
 - CQRS nos permite escalar comandos y queries de forma independiente
 - Esta es una gran ventaja en sistemas donde las lecturas superan a las escrituras.

¿Por que necesitamos CQRS?

- Separar comandos y queries nos permite optimizar ellos para alta performance.
 - Para incrementar la performance de lectura a una base de datos, uno debería crear un numero de indices. **El problema, sin embargo, es que los indices tienden a hacer mas lento las escrituras a la base de datos.**
 - Por eso, tiene mas sentido ir por un diseño optimizado de CQRS donde tenemos dos bases de datos separadas. De esta manera podemos agregar mas indices a las tablas o colecciones en la base de datos de consultas para optimizar la performance en los queries, y mientras tanto eliminamos todos los indices de las tablas o colecciones de la base de datos de comandos para optimizar las escrituras.

¿Por que necesitamos CQRS?

- Ejecutar comandos y operaciones de queries en el mismo modelo puede causar contención de datos.
- La contención de datos hace referencia a multiples procesos o instancias compitiendo por acceder al mismo indice o bloque de datos al mismo tiempo.

¿Por que necesitamos CQRS?

- Leer y escribir representaciones de datos generalmente difieren sustancialmente.
- En el lado de comandos, podría haber columnas adicionales o campos que necesitan ser actualizados durante la ejecución de un comando que no son necesarios en el lado de consultas.
- Segregando en comandos y queries, reduce la complejidad y hace a los modelos más mantenibles y flexibles.

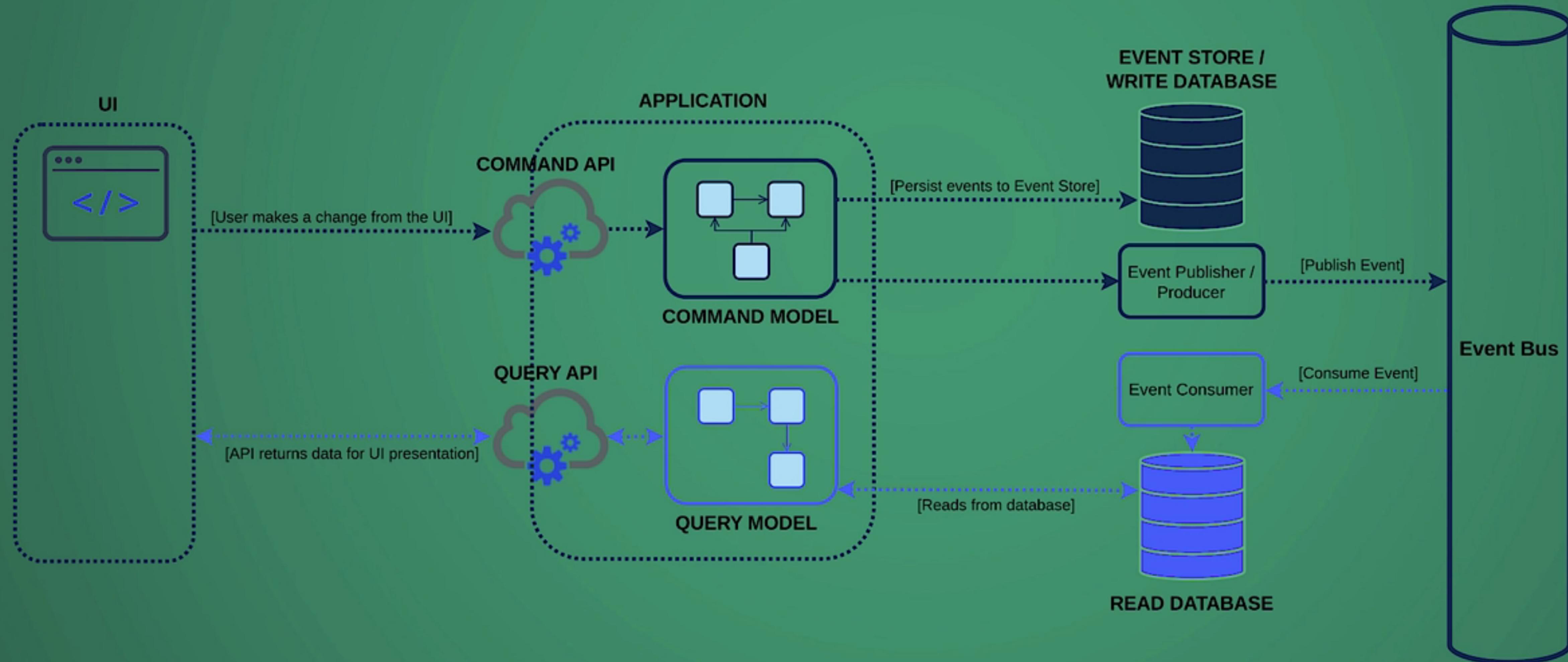
¿Por que necesitamos CQRS?

- La separación nos da la habilidad de administrar la seguridad y permisos a comando y queries de forma distinta.
- Esto podría ser por ejemplo, seguridad en la base de datos o control de acceso a usuarios. Si tu tienes dos bases de datos separadas, las lecturas no requieren acceso a la base de datos de escritura. Al mismo tiempo, tener una separación entre escrituras y lecturas simplifica los permisos de acceso entre comandos y queries.

¿Qué es Event Sourcing?

- Event Sourcing define un approach donde todos los cambios que son hechos a un objeto o entidad son almacenados como una secuencia de eventos inmutables en un almacén de eventos, en lugar de guardar solamente el estado actual.
- Cuando el estado de una entidad de negocio cambia, un nuevo evento es agregado a la lista de eventos en el almacén de eventos. El evento debe ser una simple operación, y es inherentemente atómico.
- Los eventos son hechos, que representan alguna acción que sucedió en el sistema.
- Un evento es un sistema eCommerce podría ser AddedToCart, CartUpdated, ProceededToCheckout, OrderPlaced, PaymentMade, o PaymentRejected.

CQRS + Event Sourcing



Beneficios del Event Sourcing

- El almacén de eventos nos provee un blog completo de cada cambio de estado , es decir, crea un audit trail del sistema completo
- El estado de cualquier objeto puede ser recreado siguiendo los pasos del almacén de eventos.
- Esto nos da la posibilidad de revertir el sistema a un estado previo. Esto es especialmente útil en debugging.
- Esto elimina la necesidad de mapear tablas relacionales con objetos
- Un event store puede enviar data a múltiples bases de datos de lectura
- En el caso de una falla, el event store puede ser usado para restaurar las bases de datos de lectura.

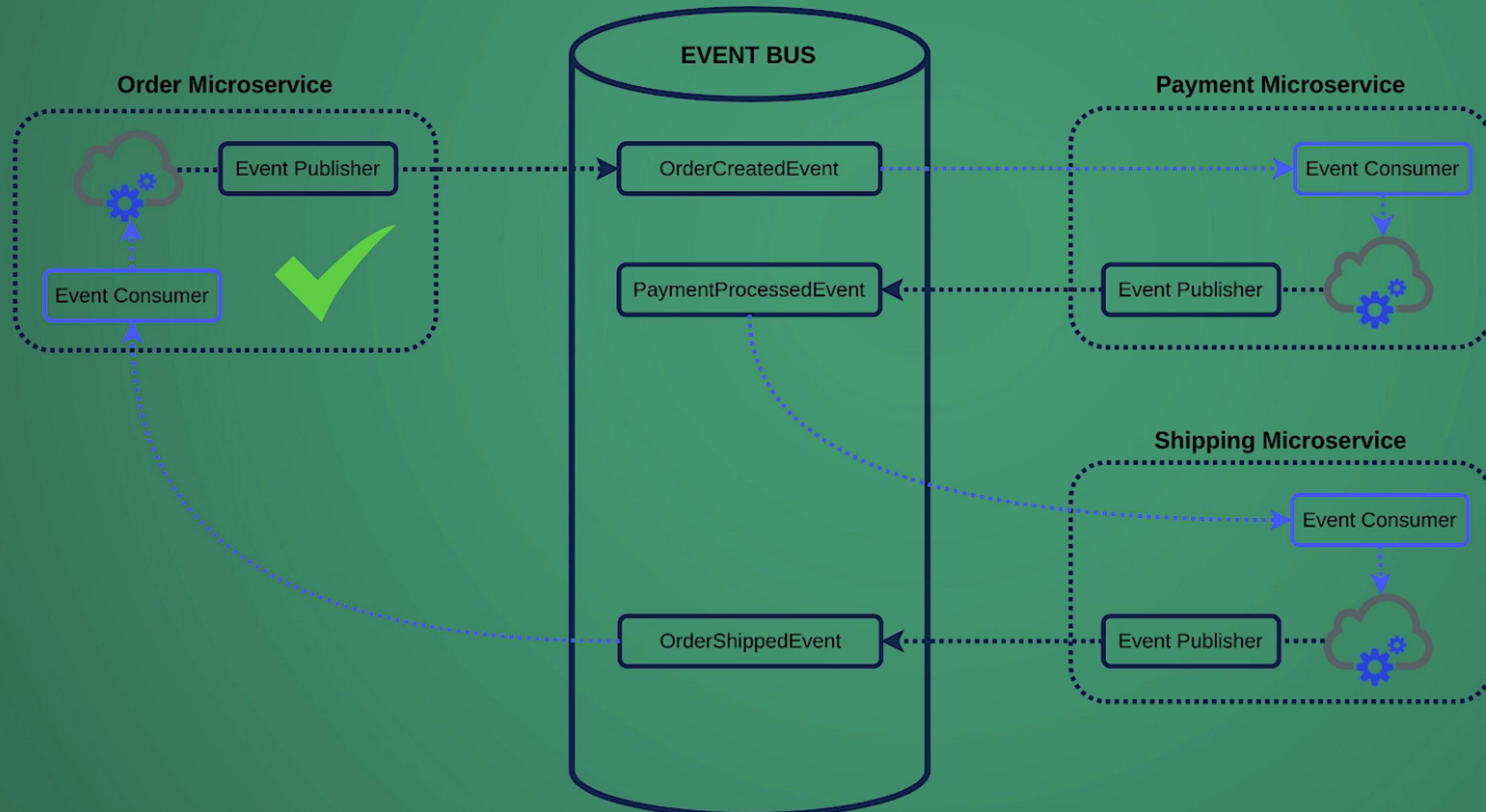
¿Qué es una transacción distribuida?

- Recordemos que el propósito de una transacción de base de datos es garantizar todo o nada y cumplir las 4 propiedades ACID (Atomicity, Consistency, Isolation, Durability).
- En simples palabras, eso significa que si cualquier paso en la transacción falla, todo en la transacción será deshecho (rolled back).
- Una transacción distribuida puede ser definida como una transacción de base de datos que involucra dos o más hosts de red (o micro servicios) como alternativa a un simple sistema que ejecuta una transacción directamente a la base de datos.

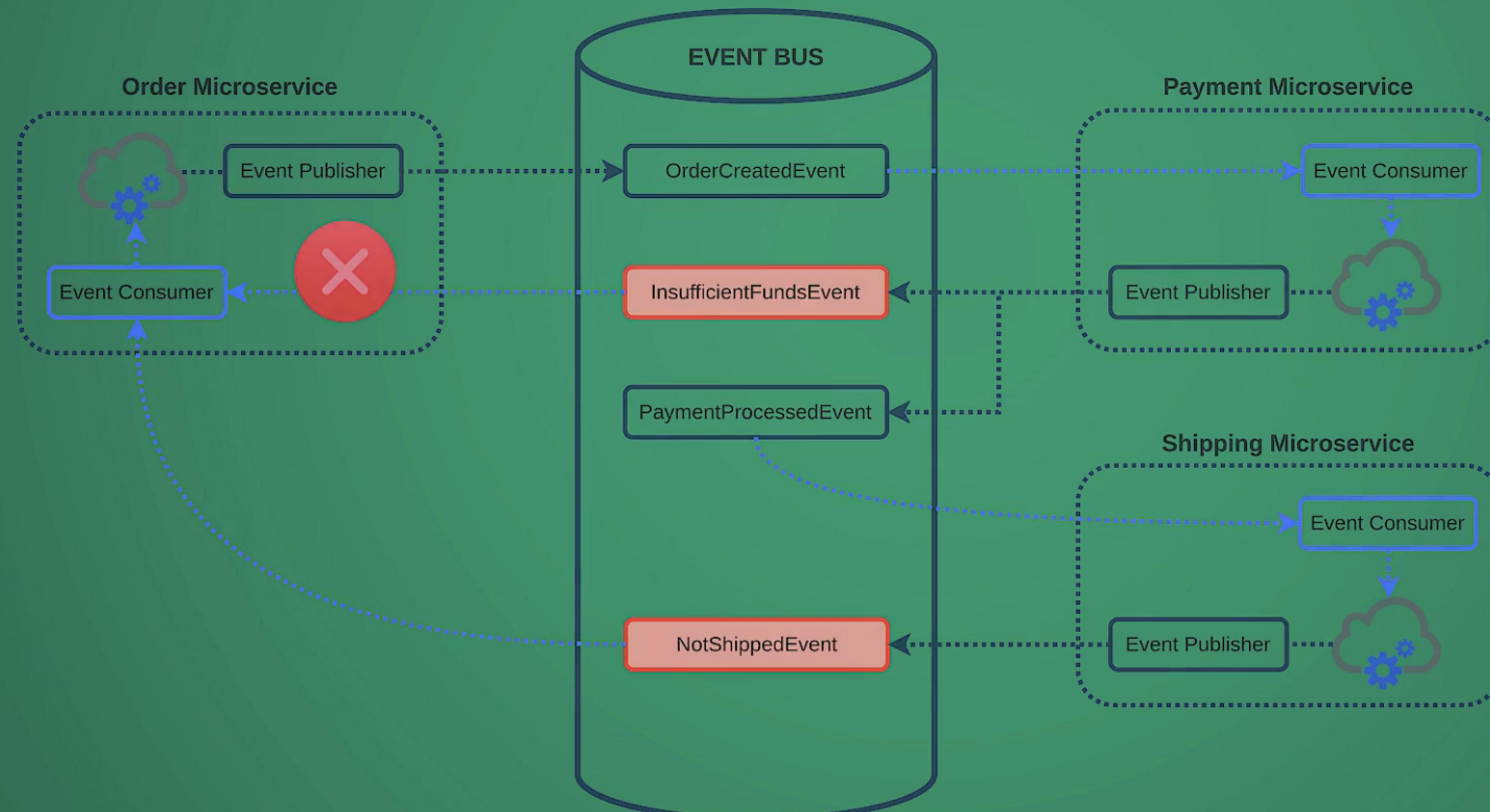
¿Qué es el patrón SAGA?

- El patrón SAGA fue introducido en 1980 como solución para transacciones distribuidas en base de datos relacionales.
- El patrón SAGA es un patrón de diseño que provee una solución para implementar transacciones en la forma de sagas que abarca dos o mas microservicios.
- Un SAGA puede ser definido como una secuencia de transacciones locales. Donde cada uno de los micro servicios participantes ejecuta una o mas transacciones locales, y luego publica un evento que es usado para disparar la siguiente transacción en un saga, que reside en otro micro servicio participante.
- Cuando una de las transacciones en la secuencia falla, el saga ejecuta una serie de transacciones de compensación para deshacer los cambios que fueron hechos por las transacciones precedentes.

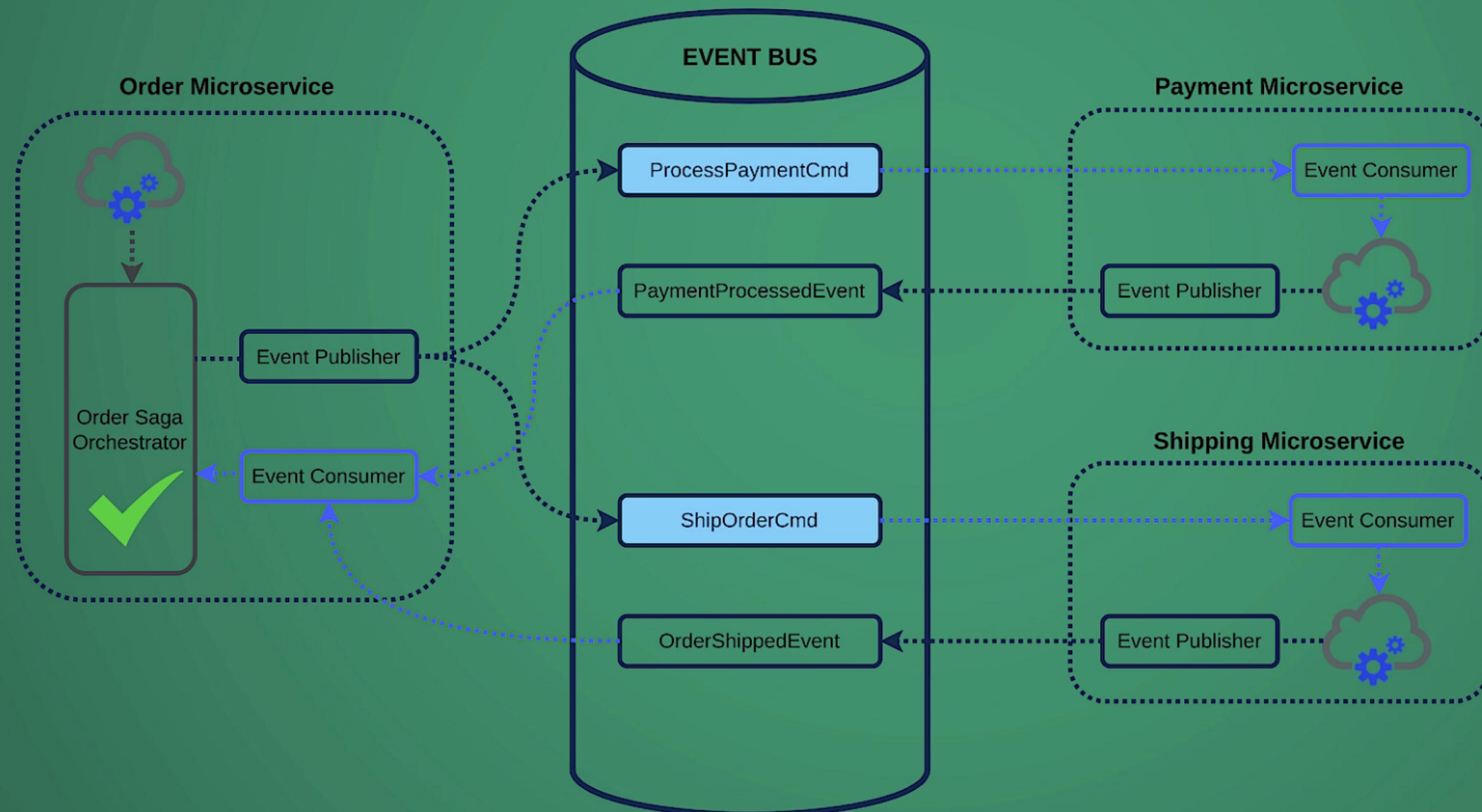
Choreography-Based Saga: Success



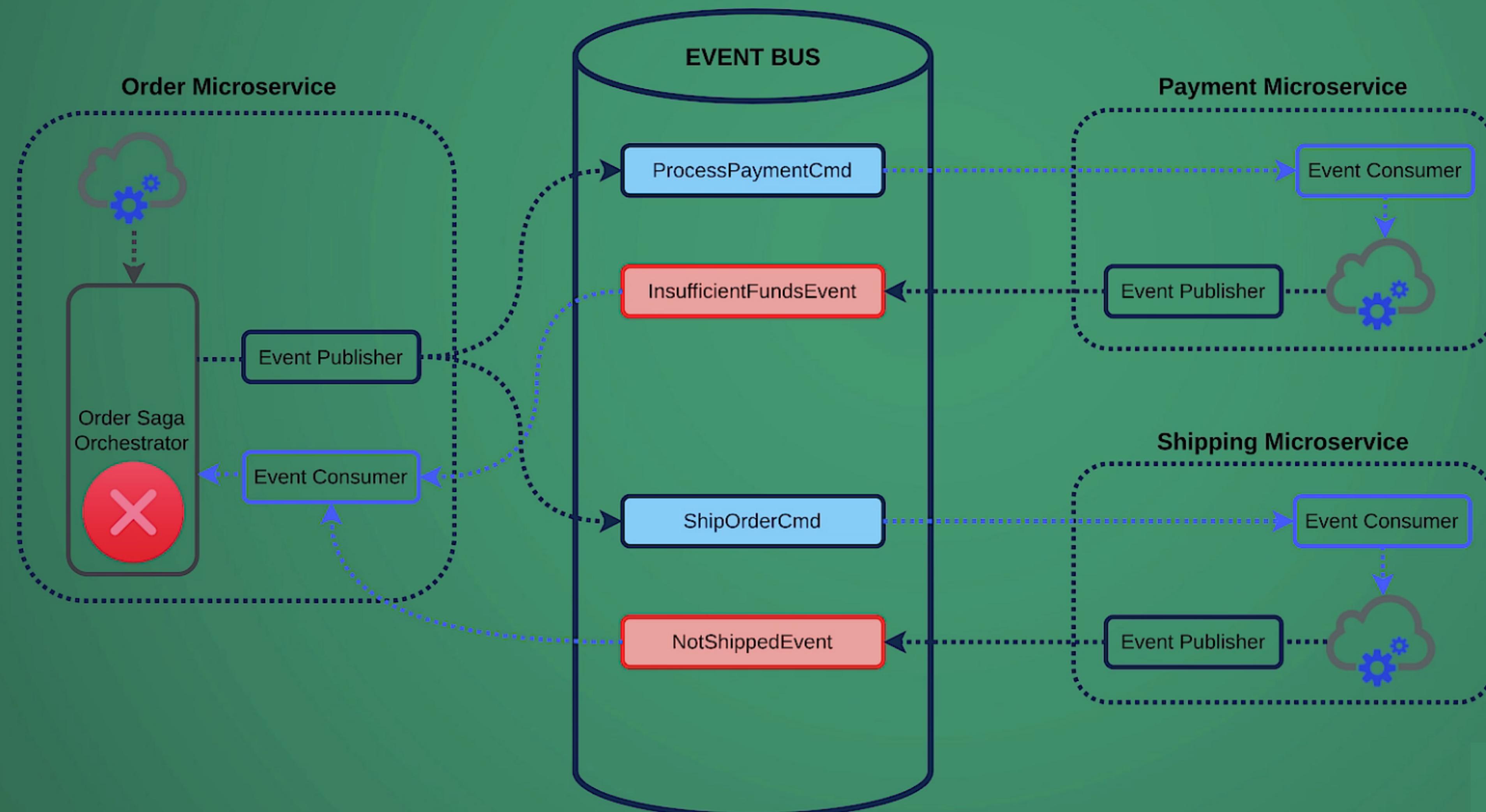
Choreography-Based Saga: Failure



Orchestration-Based Saga: Success



Orchestration-Based Saga: Failure



Factores de Exito

Factores de éxito

- Logging
- Monitoring & Alerting
- Documentation

¿Qué deberíamos loggear?

- Excepciones
- Todos los requests y responses
- Tiempos de responses de los microservicios
- Eventos que son publicados por el event bus
- Eventos que son consumidos desde el event bus
- Todos los intentos de login/access

Excepciones

- La primera forma de logging que cuquier sistema debería tener es el exception logging. Este es aun mas importante con micro servicios.
- Cuando las excepciones son loggeadas, es importante que los detalles del microservicios que produjo la excepción sean incluidos
- Especialmente cuando estas ejecutando múltiples instancias del mismo micro servicio

Todos los requests y responses

- Una de las mas efectivas formas de logging es loggear todo el contenido del request y response de un micro servicio
- Asegurate de que la información sensitiva este enmascarada cuando hagas este tipo de logging.
- Es util incluir los códigos de status HTTP cuando hagas logging de responses.

Tiempos de respuestas de los micro servicios

- Es importante monitorear cuanto tiempo toma para un micro servicio procesar un request
- Asimismo, necesitamos asegurarnos de que guardamos el timespan del request y response para poder monitorear y alertar los tiempos de respuesta excesivos.

Eventos que son publicados

- Complejas funciones de negocios pueden ser ejecutados, a través, de una cadena de micro servicios que interactúan unos con otros usando el event bus.
- Adicionalmente, es muy importante, loggear cada evento que es producido por un micro servicio, e incluir identificadores que pueden ser usado para hacer seguimiento a eventos desde el inicio hasta el final.

Eventos que son consumidos

- Deberíamos loggear cada evento que es consumido por cada consumidor.
- De esta manera podemos determinar el consumer lag. Esto es cuando tu ves que muchos eventos pueden ser producidos vs cuantos eventos pueden ser consumidos.

Intentos de login/access

- Todos los login/access token (JWT) requests que son hechos por las aplicaciones clientes deberían ser logueados.
- Esto nos da data que puede ser usada para ver si hay intentos no autorizados.
- Tambien nos dirá cuantas veces al dia, que clientes hacen muchos requests, mientras otros menos. Esto nos da insights en términos de planeamiento de escalabilidad, o para automatizar el escalamiento de micro servicios.

¿Qué deberíamos monitorear?

- Uptime de micro servicios
- El promedio de tiempo de respuesta de cada micro servicio
- Uso de recursos de cada micro servicio
- El ratio de éxito/falla de cada micro servicio
- Frecuencia de acceso de requests de clientes
- Dependencias de infraestructura

Uso de recursos

- Hay buenas herramientas de monitoreo donde puedes ver exactamente cuales micro servicios son mas usados y cuales menos.
- Esta forma de monitoreo es especialmente util para ver si el cambio de código o upgrade a un micro servicio tuvo un positivo o negativo efecto en el uso de recursos.
- Tambien nos permite calcular el promedio de uso de recursos de cada micro servicio, lo cual ayuda en el planeamiento de expansión y escalabilidad

Monitorear frecuencia de acceso

- Es muy importante monitorear la frecuencia de acceso de los requests del cliente. En una arquitectura de micro servicios, las aplicaciones clientes tienen credenciales de login que son usadas para requerir un Access token (JWT).
- Si hay un incremento de client requests, esto podría ser indicador de que tu necesitas escalar tu servicio de autenticación, API Gateway u otros micro servicios que son observados durante el incremento del client requests.
- Es también bueno para la seguridad ver si los clientes están obteniendo constantes respuestas de unauthorized de nuestros micro servicios.

Buenas practicas de Documentación

- API Documentation
- Design documentation
- Dependencies
- Network and port allocations

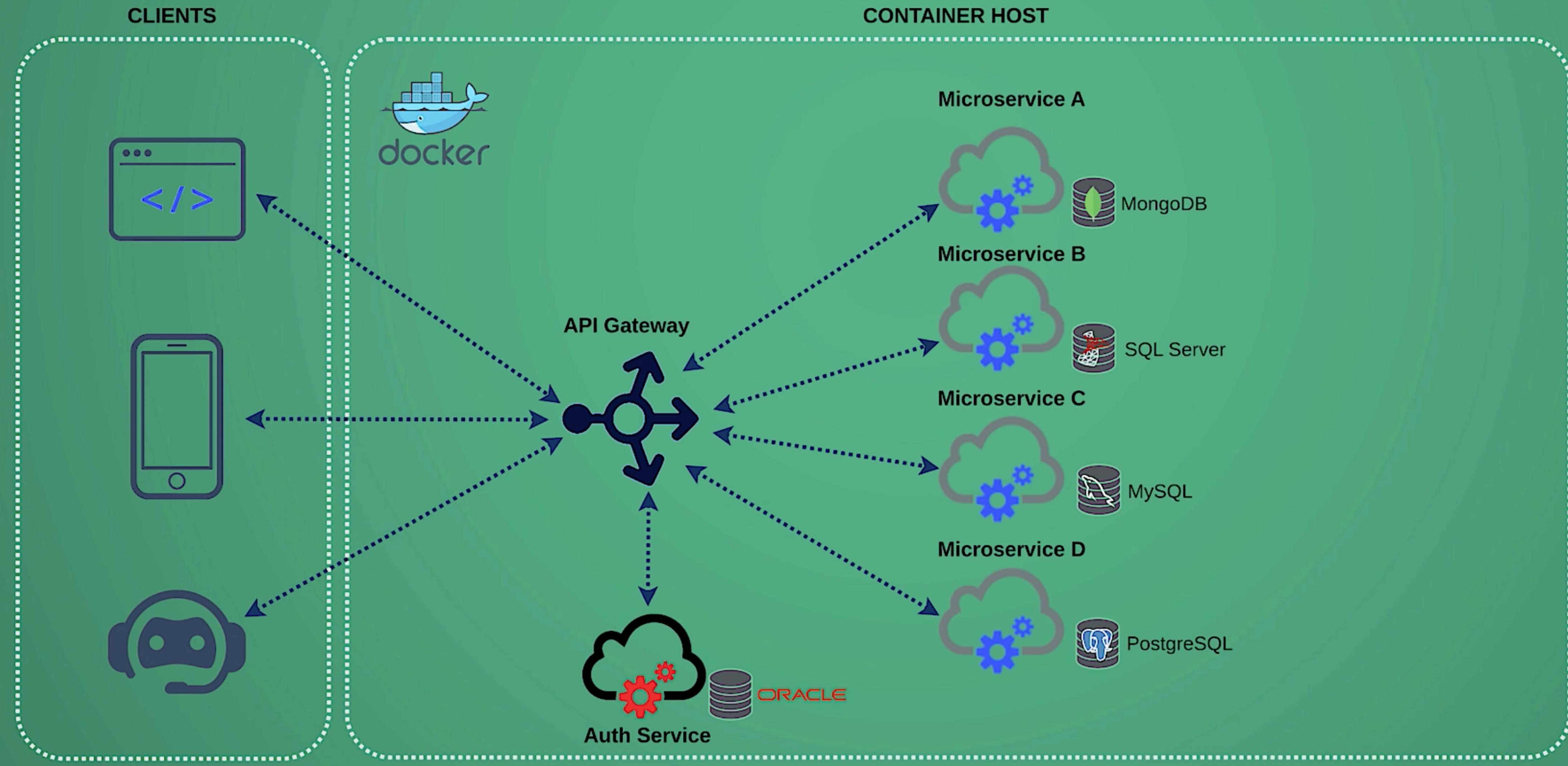
API Documentation

- Los micro servicios son usualmente creados como REST APIs, y puesto que cada micro servicio tiene un propósito específicos, es muy importante que tengamos una adecuada documentación.
- Debería proveer una descripción de cada micro servicio, e indicar el propósito de cada uno. Este debería proveer el JSON esperado en request y response para simplificar la integración con el cliente
- Finalmente, la documentación API debería ser como catalogo, donde los desarrolladores pueden fácilmente ver cada microservicio disponible y cual es el propósito de cada uno de ellos.

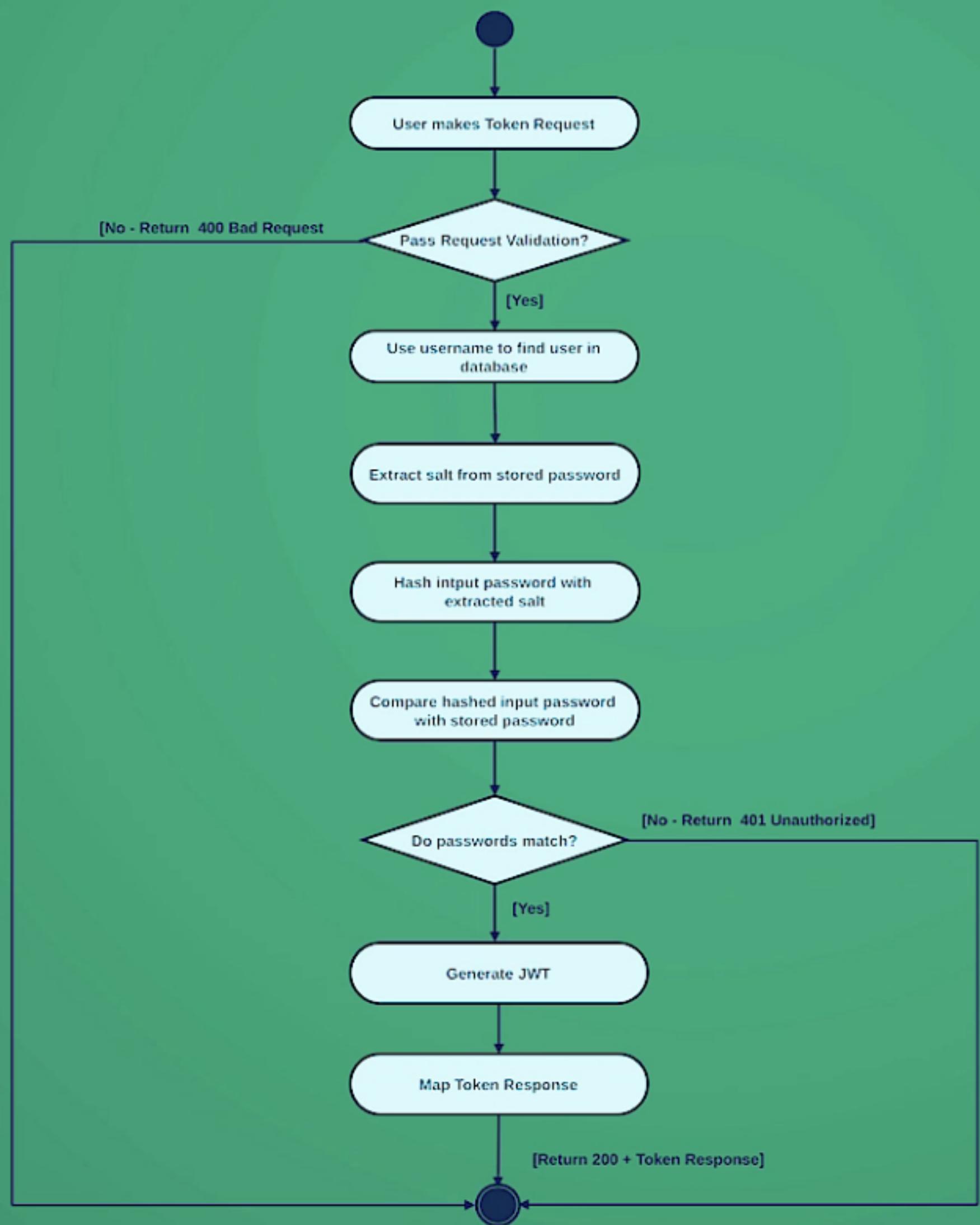
Design Documentation

- Como cada micro servicio es distribuido, la documentación de diseño es muy importante
- Hay dos tipos de diseño de documentación que se recomienda:
 - Uno para documentar como cada micro servicio forma parte de la arquitectura y otro para ilustrar la lógica de negocio de cada micro servicio
 - Ejemplo: la vista de despliegue de la arquitectura, un diagrama de actividades para ilustrar la lógica de negocio de cada micro servicio.

Typical Microservices Architecture



2. Design Documentation



Dependencies

- Es importante documentar las dependencias de micro servicios.
- Esto puede incluir librerías de terceros o paquetes del cual depende el micro servicio. También pueden ser stored procedures o vistas de una base de datos relacional. O driver mapeados en un Linux host.
- El punto es que necesitamos saber las dependencias de un microservicio para asegurarnos de que no tendremos problemas si escalamos el micro servicio. De esta manera, no nos sorprendemos cuando el micro servicio no trabaja con un nuevo host.

Network and port allocations

- Necesitamos un registro de la red y port allocations de cada micro servicio.
- Necesitamos saber que puertos están ya en uso en un host antes de desplegar un nuevo servicio.
- También considerar configurar un estándar para Port allocation. Tu podrías reservar un rango específico de puertos para un grupo de micro servicios o para herramientas de infraestructura que se ejecutan en el host
- Mantener en mente que hay puertos que el SO u herramientas comunes usan y debemos evitarlos

Despliegue e infraestructura

Herramientas y tecnologías

- Framework y lenguajes de programación
- Tecnologías de contenedores
- Engines de orquestación
- Service Discovery
- API Gateways
- Event Bus tools & technologies
- Logging tools
- Monitoring tools
- Documentation tools
- Testing tools

Tecnologías de contenedores

- Docker
- CoreOS' rkt
- LXC Linux Containers
- OpenVZ
- Containerd
- Podman

Orchestration Engines

- Kubernetes (K8s)
- Docker Swarm
- OpenShift
- Cloud Foundry's Diego
- CoreOS Fleet
- Mesosphere Marathon
- Amazon ECS, EKS
- Azure Kubernetes Service (aks)

Service Discovery

- Consul
- Apache Zookeeper
- Etcd
- Eureka
- SmartStack
- SkyDNS
- Vine
- Baker Street

API Gateways

- Kong
- Ambassador
- Ocelot
- Tyk
- Amazon AWS API Gateway
- Azure Api Management
- Spring Cloud Gateway
- KrakenD

Event Bus tools & technologies

- Apache Kafka
- RabbitMQ
- Azure Service Bus
- Amazon Simple Queue Service (SQS)
- Google cloud Pub/Sub
- Azure EventBridge

Logging Tools

- FluentD
- Graylog
- Kibana
- Logstash
- Bunyan
- Suro

Monitoring Tools

- Grafana
- Prometheus
- cAdvisor
- Riemann
- Spigo
- Sensu
- Sysdig Monitor

Documentation Tools

- Swagger UI
- Apiary
- readme.io
- Slate
- Gelato
- Aglio
- LucyBot's DocGen

Testing Tools

- Postman
- Hoverfly
- Pact
- Jmeter
- Gatling
- REST-Assured
- Citrus Framework