



# APIs (Formas de comunicación alternativas)

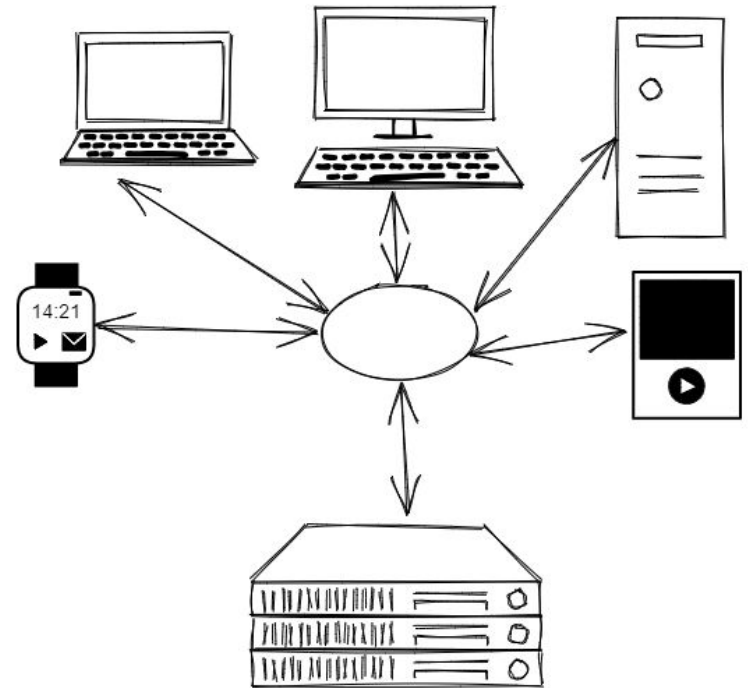


# Comunicación Cliente Servidor

Como ya hablamos la comunicación básica en un sistema se da entre dos nodos generando una comunicación “cliente servidor”.

Estos roles tradicionalmente tomados por “computadoras” ahora son tomados por **diversos dispositivos**:

- » Dispositivos móviles.
- » Sensores.
- » IoT.
- » Electrodomésticos.
- » Maquinarias.



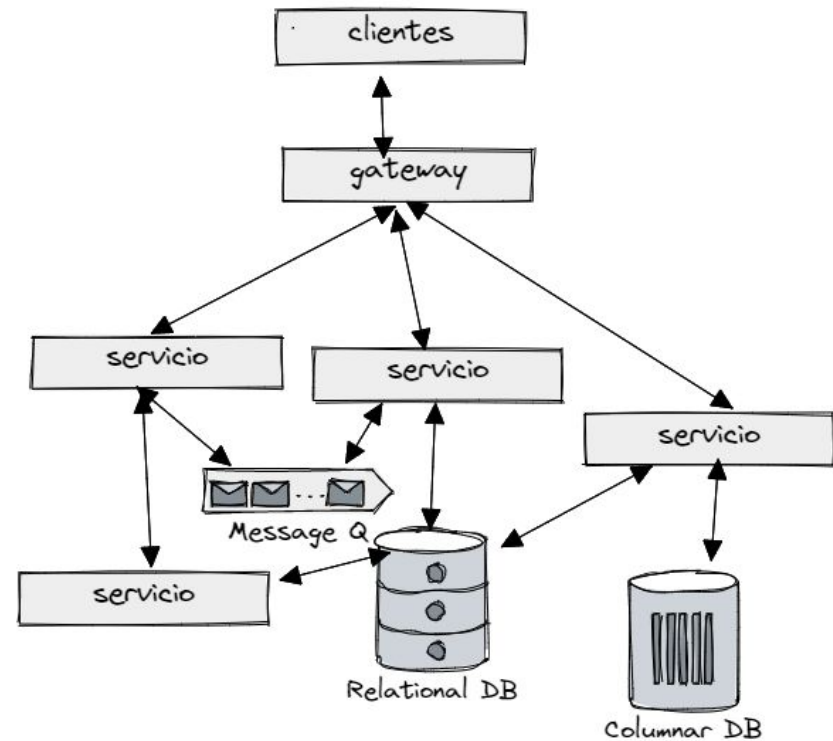
# Comunicación Cliente Servidor

Y los “servicios” se van complejizando de manera que los mismos se resuelven a través de la colaboración de diversos componentes, que se comunican entre ellos.

Esto provoca que existan diversas formas de establecer comunicación entre dos puntos de un sistema.

**gRPC** nos sirvió como ejemplo para analizar internamente una manera de comunicación.

Hoy veremos algunos de los estándares alternativos más populares para generar APIs de consumo de servicios.






# APIs

La forma de la API es el punto central a definir ya que es la que indica la semántica y las operaciones posibles a la hora de consumir un servicio.


Para implementarlas existen diversos paradigmas, estándares, frameworks, y tecnologías a elegir.





# APIs

A pesar de que cualquiera de ellas puede utilizarse para resolver casi cualquier API se pueden tomar diversos criterios para elegir la “mejor” para el servicio que estamos implementando.

- Formas de comunicación y periodicidad.
  - Transporte.
  - Diversidad de clientes, lenguajes, tecnologías.
  - Maneras de describir las operaciones.
  - Diversidad de los payloads.
- 

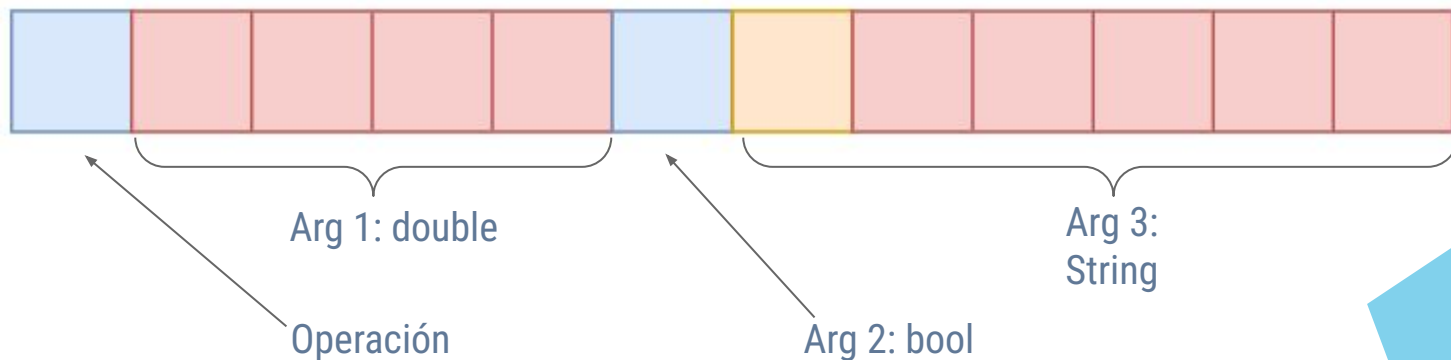


# Tira de Bytes

# Tira de Bytes

La tira de Bytes es la forma más “primitiva” y/o “básica” de generar una API.

Durante mucho tiempo fue la única opción hasta que se generaron nuevas y mejores alternativas.



Es como funcionan los protocolos como TCP. Se dice algo del estilo "en el primer byte va el numero de operacion", el resto es el payload.

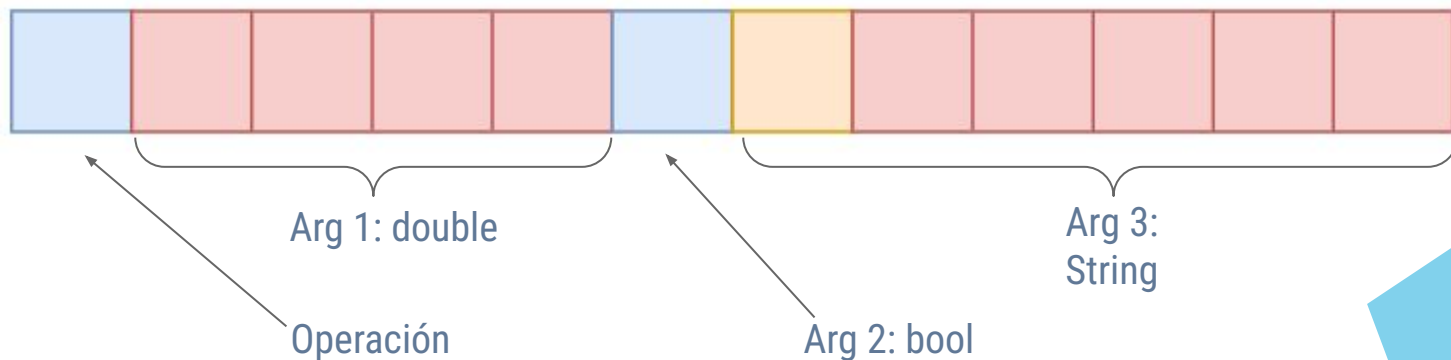
En el caso de este ejemplo:

- El primer byte es la operacion
- El double es el payload size
- 
- El string es el payload

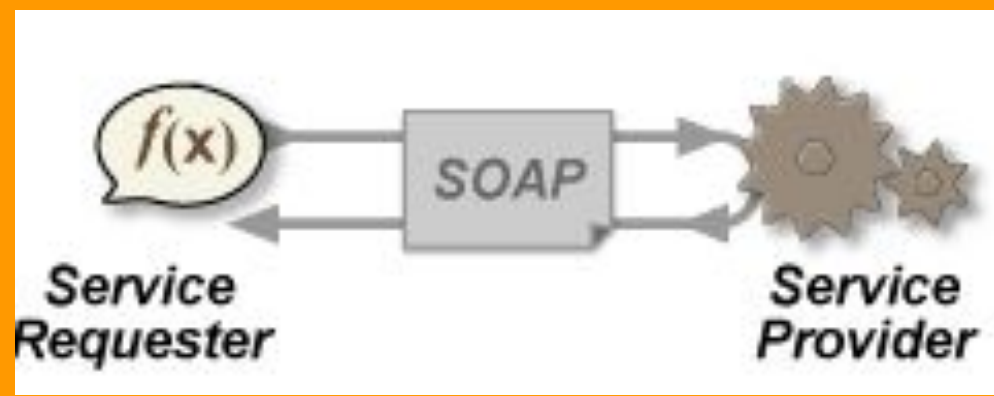
# Tira de Bytes

Pero al día de hoy todavía pueden encontrarse algunas **API legacy** que usan alguna variante de la misma.

Se trata de abrir una conexión por socket y convenir (mediante el uso de documentación) de que manera se envía los diferentes valores del request y la response.







**SOAP**



# SOAP

**SOAP** (Simple Object Access Protocol) es un protocolo estándar de comunicación mediante el **intercambio de datos utilizando XML**.



# SOAP

## Ejemplo de un mensaje en xml.

Algunas ventajas:

- Altamente customizable
- Proporciona muy buena seguridad
- Se pueden definir estructuras (en JSON no tanto)

```
<?xml version="1.0" ?>
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>


</soap:Envelope>
```



# SOAP - Características

La definición de los servicios también se hace mediante XML.

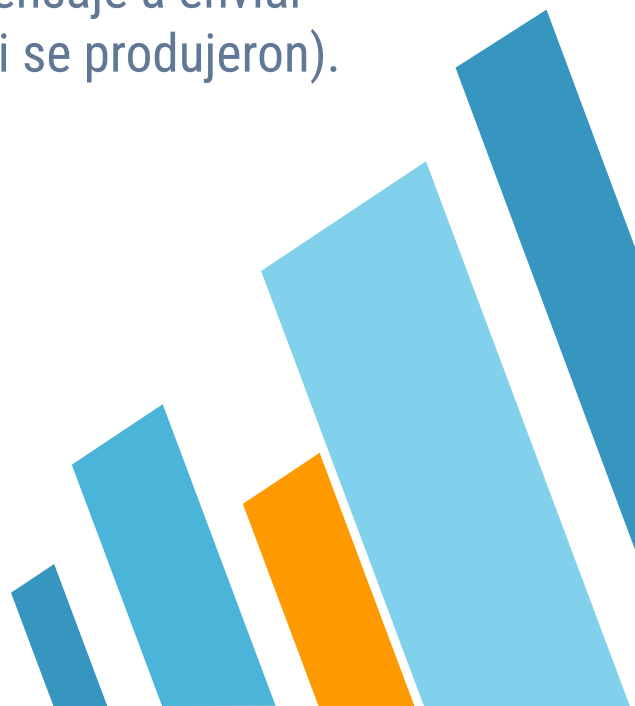
El protocolo tiene 3 características principales:

- **Extensibilidad**: El protocolo pueden extenderse para hacer servicios complejos o agregar seguridad y routing
  - **Neutralidad**: Utiliza transporte por TCP y puede aplicarse diversos protocolos de aplicación como HTTP, SMTP o JMS.
  - **Independencia**: (permite cualquier modelo de programación) al definirse vía XML se puede programar en diversos lenguajes y paradigmas.
- 



# SOAP - Mensaje

Un mensaje en SOAP consta de las siguientes partes:

- **Envelope**: Elemento que engloba
  - **Header**: Elemento opcional que tiene atributos que son complementarios al mensaje, por ejemplo, elementos de autenticación.
  - **Body**: Elemento obligatorio con el contenido del mensaje a enviar
  - **Fault**: Información de errores que se produjeron (si se produjeron).
- 

# SOAP - Mensaje

Esto va todo por HTTP POST, no se usa otro verbo que no sea POST

Ejemplo de la estructura de un mensaje.

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV =
"http://www.w3.org/2001/12/soap-envelope"
      SOAP-ENV:encodingStyle =
"http://www.w3.org/2001/12/soap-encoding">

  <SOAP-ENV:Header>
    ...
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    ...
    <SOAP-ENV:Fault>
      ...
    </SOAP-ENV:Fault>
    ...
  </SOAP-ENV:Body>
</SOAP_ENV:Envelope>
```

# SOAP - Definición de un servicio

La definición de los mensajes, puertos y operaciones.

```
<wsdl:message name="getMessageRequest">
  <wsdl:part element="impl:getMessage" name="parameters">
  </wsdl:part>
</wsdl:message>

<wsdl:message name="getMessageResponse">
  <wsdl:part element="impl:getMessageResponse"
    name="parameters"></wsdl:part>
</wsdl:message>

<wsdl:portType name="Greetings">
  <wsdl:operation name="getMessage">
    <wsdl:input message="impl:getMessageRequest"
      name="getMessageRequest">
    </wsdl:input>
    <wsdl:output message="impl:getMessageResponse"
      name="getMessageResponse">
    </wsdl:output>
  </wsdl:operation>
</wsdl:portType>
```

# SOAP - Definición de un servicio

WSDL es el formato en XML con el cual se define el servicio

```
<wsdl:types>
  <schema elementFormDefault="qualified"
targetNamespace="http://DefaultNamespace"
xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="getMessage">
        <complexType>
            <sequence>
                <element name="message" type="xsd:string"/>
            </sequence>
        </complexType>
    </element>
    <element name="getMessageResponse">
        <complexType>
            <sequence>
                <element name="getMessageReturn"
                    type="xsd:string"/>
            </sequence>
        </complexType>
    </element>
  </schema>
</wsdl:types>
```



# SOAP - Definición de un servicio

Y la definición de los bindings y el servicio.

```
<wsdl:binding name="GreetingsSoapBinding" type="impl:Greetings">
  <wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getMessage">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getMessageRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="getMessageResponse">
      <wsdlsoap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="GreetingsService">
  <wsdl:port binding="impl:GreetingsSoapBinding" name="Greetings">
    <wsdlsoap:address
location="http://localhost:8080/WebServiceSoap/services/Greetings"/>
  </wsdl:port>
</wsdl:service>
```



# SOAP - Generar código

Los lenguajes proveen de **compiladores** que permiten **interpretar las definiciones del xml** y **generar las clases de servicios** y stubs para que luego se pueda completar la implementación del servicio.

El proceso es similar al visto con gRPC, varían las clases a implementar y los nombres.

En **Java** existen **librerías** como **CXF** (más popular), **AXIs** y **soporte nativo** vía JAX-WS. La necesidad de usar uno u otro puede estar dada por el código existente.

Estas librerías generan las clases Java de la misma manera que los .proto generan Java.






## SOAP - Ventajas e inconvenientes

Durante mucho tiempo SOAP fue el sistema de web services más popular debido a sus características de extensibilidad y multi lenguaje.


Las **contra** que tiene son que la definición via **xml**, puede ser **muy “verbosa”**, que la definición de servicios puede ser compleja (aún para servicios simples).





## SOAP - Ventajas e inconvenientes

Entonces fue **perdiendo popularidad contra REST** y se generó una dicotomía donde **REST se usa para servicios “simples” web y B2C;** y **SOAP se usa para aquellos más complejos con necesidades de mayor seguridad como Financieras y B2B**





**Rest**




# Rest

Según la definición original escrita en la tesis doctoral de Roy T. Fielding REST (representational state transfer) es:

**“Estilo arquitectónico para sistemas distribuidos de hipermedia, describiendo los principios de ingeniería de software que guían REST y las constantes de interacción elegidas para retener esos principios”**

Entonces originalmente era un conjunto de principios de arquitectura. Como Fielding trabajó en el protocolo HTTP y los principios de REST están muy atados a ese protocolo el REST tomó un significado más amplio.





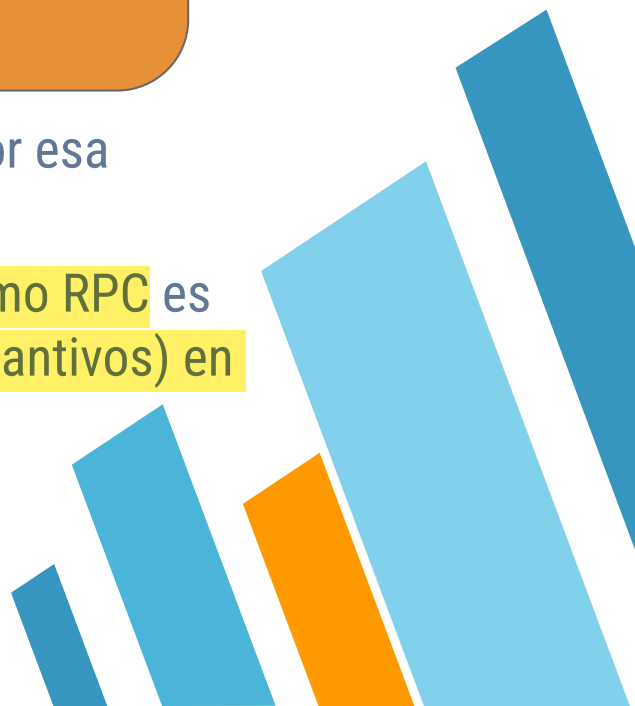
# Rest

Entonces en estos tiempos se podría describir a Rest no oficialmente como

Una **interfaz remota de servicios basada en Recursos sobre el protocolo HTTP**. Esta interfaz se utiliza para obtener y modificar datos multimedia y texto en diversos formatos como JSON, XML, binario, etc.


Rest probablemente sea la interfaz web más popular por esa conexión con HTTP.

La **principal diferencia con otros tipos de interfaces como RPC** es la **definición de los servicios a través de Recursos (sustantivos) en vez de Verbos (acciones)**.






# Rest - Principios de Arquitectura

1. **Arquitectura cliente-servidor (Client-Server):** Para lograr separación de responsabilidades y desacoplar la interacción entre el cliente y el servidor. Como la dependencia es a través del mensaje modificar cada componente no debería afectar al otro.
- 






# Rest - Principios de Arquitectura

2. **Ausencia de estado (Stateless)**: El estado/contexto de la petición es mantenida por el cliente. El servidor es stateless y recibe todo del cliente. Esto va a permitir que diversas instancias del servicio puedan atender llamadas sucesivas de un mismo cliente.
- 



# Rest - Principios de Arquitectura

3. **Habilitación y uso del cache (Cacheable):** Todos los recursos deben ser cacheables, mientras y por el tiempo que tenga sentido. Esto permite optimizar la comunicación entre el cliente y el servidor
- 




# Rest - Principios de Arquitectura

4. **Sistema por capas (Layered):** Un cliente debe conocer únicamente la capa a la que le está hablando. Es decir, no debe tener en cuenta aspectos concretos



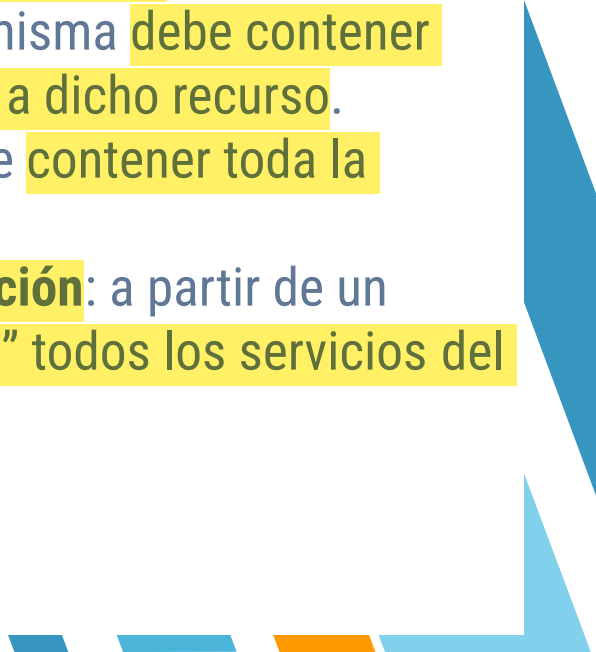


# Rest - Principios de Arquitectura

5. **Ejecución de código (Code on Demand)**: Esto era **opcional** y con el tiempo entró en **desuso**. Era la posibilidad de poder **ejecutar código en el cliente** (o desde el cliente) mediante el uso de scripts o applets
- 



# Rest - Principios de Arquitectura


6. **Interfaz uniforme**: se aplica el principio de “generalidad” para simplificar las interacciones. Para ello define las siguientes restricciones:
    - a. **Identificación de recursos en las peticiones**: Todos los recursos se identifican unívocamente. Esto es independiente de cual es la representación que se envía (que se puede solicitar) por medio de headers.
    - b. **Manipulación de recursos a través de representaciones**: Cuando el cliente obtiene la información de un recurso la misma debe contener toda la información para poder modificar/borrar a dicho recurso.
    - c. **Mensajes auto-descriptivos**: cada mensaje debe contener toda la información necesaria para que se ejecute.
    - d. **Hipermedia como motor del estado de la aplicación**: a partir de un pedido inicial el cliente debería poder “descubrir” todos los servicios del sistema
- 



# Rest - Recursos

TODOS LOS RECURSOS EN REST SON SUSTANTIVOS


**Los recursos son la información que diferentes aplicaciones proporcionan a sus clientes. Los recursos pueden ser imágenes, videos, texto, números o cualquier tipo de datos.**





# Rest - Recursos

Los recursos pueden ser de 3 tipos:

- **colección**: un conjunto de elementos que tienen características similares. Se identifican con el sustantivo que la describe (generalmente plural).
  - **singleton/documento**: una instancia individual de una colección. Se identifica mediante un identificador natural o interno y contiene el resto de los datos que contienen.
  - **subcolecciones**: son **conjuntos de singletons** que están subordinados a un elemento.
- 



# Rest - Recursos - URIs

Para acceder a los recursos se utilizan URIs.


Partiendo de la uri del servicio y potencialmente un contexto y versionado, por ejemplo: <http://remote-service/context/v1>

Jerárquicamente se agrega componentes

- **colecciones**: con el nombre del recurso: **/movies/**
- **singleton**: con la colección y el identificador: **/movies/{id}**
- **subcolección**: agregamos la colección: **/movies/{id}/actors**

En la vida real, sería un problema si quisiera listar todos los actores.

Esto tendría que ser en el endpoint **/actors**, pero esto está mal según REST porque no se puede devolver el mismo recurso desde dos endpoints.








## Rest - Recursos - URIs

Se utilizan las **features de HTTP** para hacer más específicos el pedido:

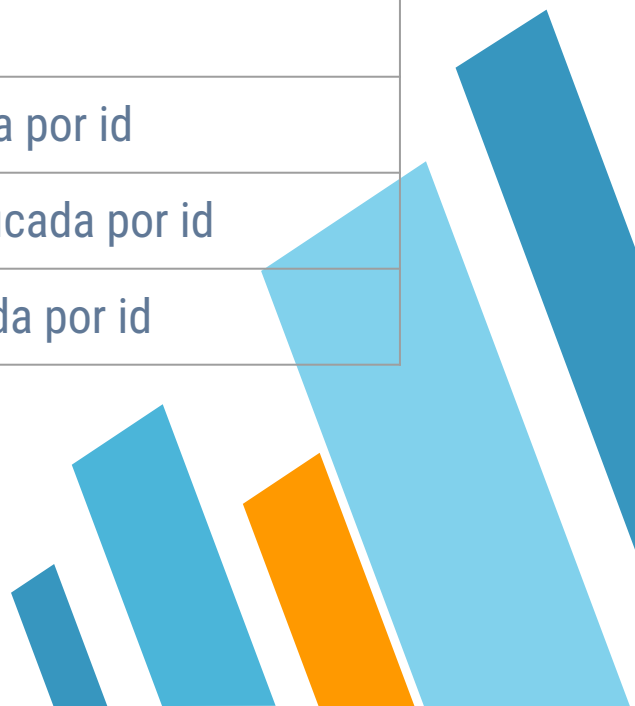
- **status codes**: para indicar **resultados**.
  - **content-types**: para pedir/indicar la **representación** a usar en el recurso
  - **query params**: para agregar **limitantes** como **criterios de búsqueda**, **paginación**, etc.
- 



# Rest - Recursos - Métodos

Los métodos HTTP permiten realizar acciones sobre los recursos como búsquedas y las operaciones CRUD.

<b>GET</b>	<b>/movies</b>	Trae todas las películas
<b>POST</b>	<b>/movies</b>	Agrega una pelicula
<b>GET</b>	<b>/movies/{id}</b>	Trae la película identificada por id
<b>PUT</b>	<b>/movies/{id}</b>	Modifica la película identificada por id
<b>DELETE</b>	<b>/movies/{id}</b>	Borra la película identificada por id



# Rest - Recursos - HATEOAS

**Hipermedia como motor del estado de la aplicación** es una de las restricciones de REST.

Básicamente indica que cada cliente debería poder navegar el servicio mediante la respuesta al primer Request. La manera de navegar es agregar a la respuesta hipermedia links a los recursos dependientes (subcolecciones o singletons).


```
{
  "movieID": 101001,
  "movieName": "Nueve Reinas",
  "year": 2000,
  "directorId": 4040,
  "links": [
    { "href": "101001/reviews", "rel": "reviews", "type": "GET" },
    { "href": "directors/4040", "rel": "director", "type": "GET" },
    { "href": "101001", "rel": "delete", "type": "DELETE" }
  ]
}
```



## Rest - Código

La mayoría de los lenguajes tienen frameworks que funcionan como middleware para poder crear aplicaciones REST simples.

Dentro del mundo Java tenemos varios, los ejemplos presentados a continuación combinan elementos de **Spring - Jersey - JAX-RS** para poder proveer un servicio que retorna un recurso llamado message representado en JSON





# Rest - Código - Objeto

Definición del objeto que representa al Recurso, en este caso un mensaje

```
public record Message(String message, LocalDateTime date) {  
  
    public Message(String message) {  
        this(message, LocalDateTime.now());  
    }  
}
```

# Rest - Código - Controlador

Definición del componente que responde los pedidos HTTP

```
@Component
@Path("/messages")
public class MessageResource {

    @Autowired
    private MessageService messageService;

    @POST
    @Produces("application/json")
    public Message message(@QueryParam("message") String
message) {
        return messageService.create(message);
    }
}
```


Este autowired en realidad no va, las cosas Autowired se ponen en el constructor. Un error que puede ocurrir si no hacemos esto es que tenemos un momento (cuando se ejecuto el constructor pero no el autowired) en el cual tenemos un objeto incompleto.



# Rest - Código - Configuración

Configuración para registrar el controlador en el framework Jersey

```
@Component
public class JerseyConfig extends
ResourceConfig {
    public JerseyConfig() {
        register(MessageResource.class);
    }
}
```



# Rest - Código

Se define una aplicación de Spring Boot con un bean servicio del tipo **MessageService**

```
@SpringBootApplication
public class RestTestApplication {
    @Bean
    MessageService buildMessageService() {
        return new DefaultMessageService();
    }

    public static void main(String[] args) {
        SpringApplication.run(RestTestApplication.class, args);
    }
}
```






## Rest - Ventajas

REST al ser uno de los estándares más populares tiene un gran soporte en lenguajes, frameworks y librerías. Además de conocimiento colectivo entre la comunidad de desarrollo:


Por las restricciones que lo definen está armado para que sea:

- **escalable**: stateless, client-server, cacheable
  - **flexible**: client-server y las formas de las representaciones
  - **independiente de la tecnología**: como el transporte es http tanto cliente y servidor pueden estar escrito en lenguajes diferentes
- 



## Rest - Desventajas

En contrapartida algunas desventajas de REST puede ser

- Al no haber un estándar definido surgen inconvenientes cuando se quieren describir sistemas o partes que no son fácilmente describibles a través de sustantivos (y/o operaciones CRUD).
  - En ciertas situaciones se vuelve incómodo cuando diferentes partes de un sistema requieren diferentes facetas de un recurso, lo cual provoca que haya varios endpoints o uno que devuelva demasiada información para atender todos los casos.
- 




GraphQL

**GraphQL**



# GraphQL

**GraphQL es  
un lenguaje de consulta y manipulación de datos  
para APIs,  
y un entorno de ejecución para realizar consultas  
con datos existentes**





# GraphQL

GraphQL permite definir APIs Web

- Con un esquema tipado
- Agnóstico del transporte y la fuente de datos.
- Explorable y descubrible.
- Con operaciones de:
  - búsqueda (queries)
  - modificación (mutation)
  - subscripción (subscription).
- Con la posibilidad de que el cliente seleccione lo que quiere recibir.



Más allá del nombre no es un lenguaje para recorrer grafos.

# GraphQL - Ventajas

Como ventajas no evidentes al compararlo con REST se destacan:

- Se pueden reducir los llamados al servidor para obtener los mismos datos
- Se pueden solicitar sólo los datos que se precisan
- Esos datos pueden provenir de múltiples fuentes.

```
"recentPosts": [  
  {  
    "id": "Post96",  
    "title": "Post 9:6",  
    "category": "Post 6 + by author 9",  
    "text": "Post category",  
    "author": {  
      "id": "Author9",  
      "name": "Author 9"  
    }  
  }  
]
```




# GraphQL - Esquema

En GraphQL la API se debe definir mediante un esquema.

Este esquema va a ser fuertemente tipado y va a definir los objetos que definen argumentos y respuestas otorgándole un estilo similar a una API RPC.

El esquema se puede publicar en el servicio y a partir del mismo el cliente puede inferir las operaciones.

El discovery y la inferencia son fáciles porque el servicio se publica en una url y todas las operaciones se ejecutan en ese endpoint con un request POST variando el body de acuerdo al parámetro.



# GraphQL - Tipos

El esquema en GraphQL consta de tipos

```
type Author {  
  id: ID!  
  name: String!  
  posts: [Post]!  
  thumbnail: String  
}
```

```
type Post {  
  author: Author!  
  category: String  
  id: ID!  
  text: String!  
  title: String!  
}
```

donde:

- Cada campo debe tener un nombre y un tipo
- Los tipos pueden ser **String, Int, Float, Boolean, and ID**
- **!** indica que el campo es obligatorio (non null)
- **[]** indica que se devuelve un array de valores.



# GraphQL - Operaciones

Para las operaciones existen tipos especiales para las diferentes operaciones. Entonces se puede definir un type Query (obligatorio), un type Mutation y una Subscription. Cada uno de ellos tienen métodos en vez de campos. Opcionalmente se los puede wrappear en un schema

```
schema {  
  query: Query  
  mutation: Mutation  
}  
  
type Query {  
  ...  
}  
  
type Mutation {  
  ...  
}
```



# GraphQL - Query

El tipo Query tiene todos los métodos que realizan consultas a los datos. Cada método tiene nombre, parámetros tipados y tipos de respuesta.

```
type Query {  
  recentPosts(count: Int, offset: Int): [Post]!  
}
```



# GraphQL - Hacer una Query

A la hora de realizar un llamado a una query el body se prefija con la keyword query opcionalmente se define parámetros para la llamada y se pasa un objeto con el nombre del objeto a llamar los parámetros y campos que se espera en la respuesta

```
query myRecentPosts($count: Int, $offset: Int)
{
  recentPosts(count: $count, offset:
    $offset) {
    id
    title
    text
    category,
    author {
      id
      posts{
        id
      }
    }
  }
}
```

En caso de parametrizar el llamado los argumentos se pasan en un objeto aparte

```
{
  "count": 2,
  "offset": 2
}
```

# Ejercicio 1


- Bajar el proyecto graphql-intro
- Analizar el contenido
- Realizar la query de los posts pidiendo el id y el titulo.



# GraphQL - Query Resolver

Para poder responder las consultas del cliente hay que implementar algún elemento que pueda responder.

En GraphQL esos elementos se llaman resolvers hay diversos tipos de resolvers de acuerdo a lo que resuelven (Query, Mutation, Subscription, Field) y eso es lo que permite tener granularidad a la hora de completar los campos pedidos con datos que provienen de diversas fuentes.






# GraphQL - Query Resolver - Código

La generación de objetos de modelo y resolvers depende mucho del lenguaje y la librería que se esté utilizando.

Existen los que son solo Middleware, los que generan código a partir del esquema, los que generan el esquema a partir del código y anotaciones, etc.



# GraphQL - Query Resolver - Java (Spring)

Existen diversas implementaciones de GraphQL en Java, particularmente en estos ejemplos usamos la de Spring que wrappea la oficial de GraphQL.

Entonces para definir un resolver definimos un **Controller** que contiene a un método anotado como **QueryMapping**.

```
@Controller
public class PostController {

    @QueryMapping
    public List<Post> recentPosts(@Argument int count,
    @Argument int offset) {
        return ...;
    }
}
```

## Ejercicio 2

- Implementar el query resolver.
- Probar la query planteada en el ejercicio anterior.
- Agregar pedir el autor






# GraphQL - Field Resolver

En Java Spring para resolver un campo utilizamos la annotation **SchemaMapping** para algún método que esté dentro de algún **Controller**.

```
@SchemaMapping(typeName = "Post", field = "author")
public Author getAuthor(Post post) {
    return ...;
}
```



# GraphQL - Field Resolver

Los Field Resolvers permiten resolver de que manera se obtiene el contenido de algún campo dentro de un tipo de respuesta de una query.

Se lo puede hacer tan granular como para decir “para este campo de este tipo” se resuelve con con este resolver.

Esto es lo que permite devolver una respuesta que se compone con información de diversas fuentes

```
@SchemaMapping(typeName = "Post", field = "author")  
public Author getAuthor(Post post) {  
    return ...;  
}
```



## Ejercicio 3

- Implementar el field resolver para el autor del post.
- 



# GraphQL - Batch Loaders

Batch Loaders es un concepto que se crea para mitigar el riesgo de caer en el problema de  $n + 1$  queries.

Este problema se da cuando yo solicito un listado de elementos que requiere para resolver parte de su contenido hacer una query más por cada elemento del sistema.

En nuestro caso por cada post necesitamos “resolver” el autor, esto requiere que se llame al field resolver 1 vez por cada post.

Obviamente tener una resolución con comportamiento lineal no es óptimo





# GraphQL - Batch Loaders

Al registrarle un Batch Loader a un Field lo que hace GraphQL es acumular los pedidos a ese campo (que generalmente es por id) y en vez de hacer n llamados con 1 elemento, hace 1 llamado con los n elementos en un array.

De esta manera quien implemente puede resolver la respuesta de una manera más eficiente



# GraphQL - Batch Loaders

En java spring la manera más simple de implementar un BatchLoader es anotar un método como **BatchMapping** e implementarlo viendo que el parámetro es una lista de los elementos “padre” y la respuesta es un Mono<Map<Pader, Hijo>>

```
@BatchMapping
public Mono<Map<Post, Author>> author(List<Post> posts) {
    Map<Post, Author> authors = new HashMap<>();

    return Mono.just(authors);
}
```

## Ejercicio 4

- Implementar el batch loading para los autores de los posts/
- Probar que se cumple el llamado optimizado.



# GraphQL - Mutations

Una mutación se define de manera similar a una query.

```
type Mutation {  
  createPost(authorId: String!, category: String, text:  
String!, title: String!): Post!  
}
```



# GraphQL - Mutations

En la consulta, también similar a la query el body pasa los argumentos de creación y define los campos de la respuesta que espera

```
mutation createPost(  
  $title: String!  
  $text: String!  
  $category: String  
  $authorId: String!  
) {  
  createPost(  
    title: $title  
    text: $text  
    category: $category  
    authorId: $authorId  
  ) {  
    id  
    title  
    text  
    category  
  }  
}
```

```
{  
  "title": "Make your own  
budget",  
  "text": "Do it at the  
beggining of the month",  
  "category": "self economy",  
  "authorId": "Author1"  
}
```

# GraphQL - Mutation Resolver

Para implementar el resolver para la mutación anotar el método con **@MutationMapping**

```
@MutationMapping
public Post createPost(@Argument String title, @Argument String text,
                       @Argument String category, @Argument String
authorId) {

    Post post = new Post(
        UUID.randomUUID().toString(),
        title,
        text,
        category,
        authorId);

    postDao.savePost(post);

    return post;
}
```



## Ejercicio 5

- Implemetar la mutación para los posts.




# APIs de Eventos



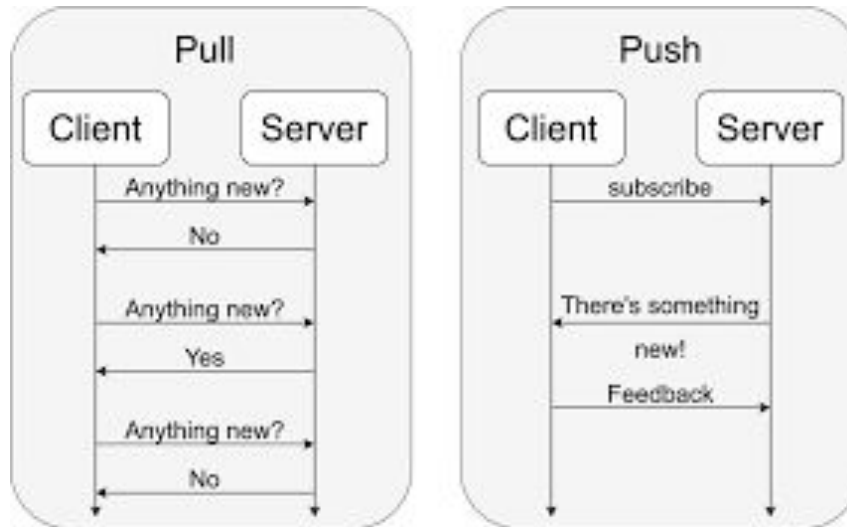
# Event Driven

Las Apis Event first o Event driven donde la comunicación de elementos relevantes al Cliente se inicia al momento de que se produce el Evento relevante



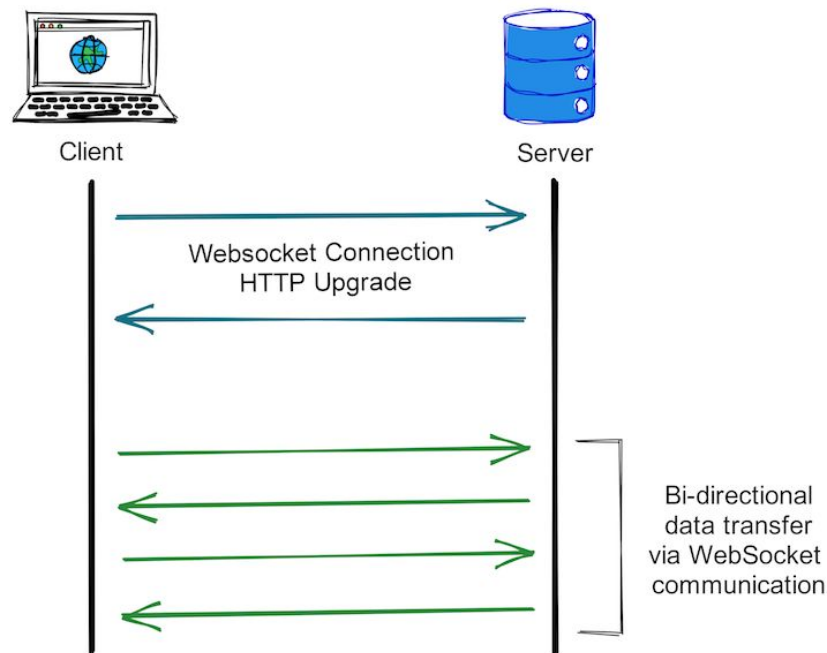
# Web Hooks

Web hooks es una manera de proveer servicios donde el cliente se comunica con el servidor indicando que "quieren consumir x eventos" y registra un "lugar donde enviarlos" cuando se generen. O sea provee un "servicio" donde el servidor publicará los eventos.



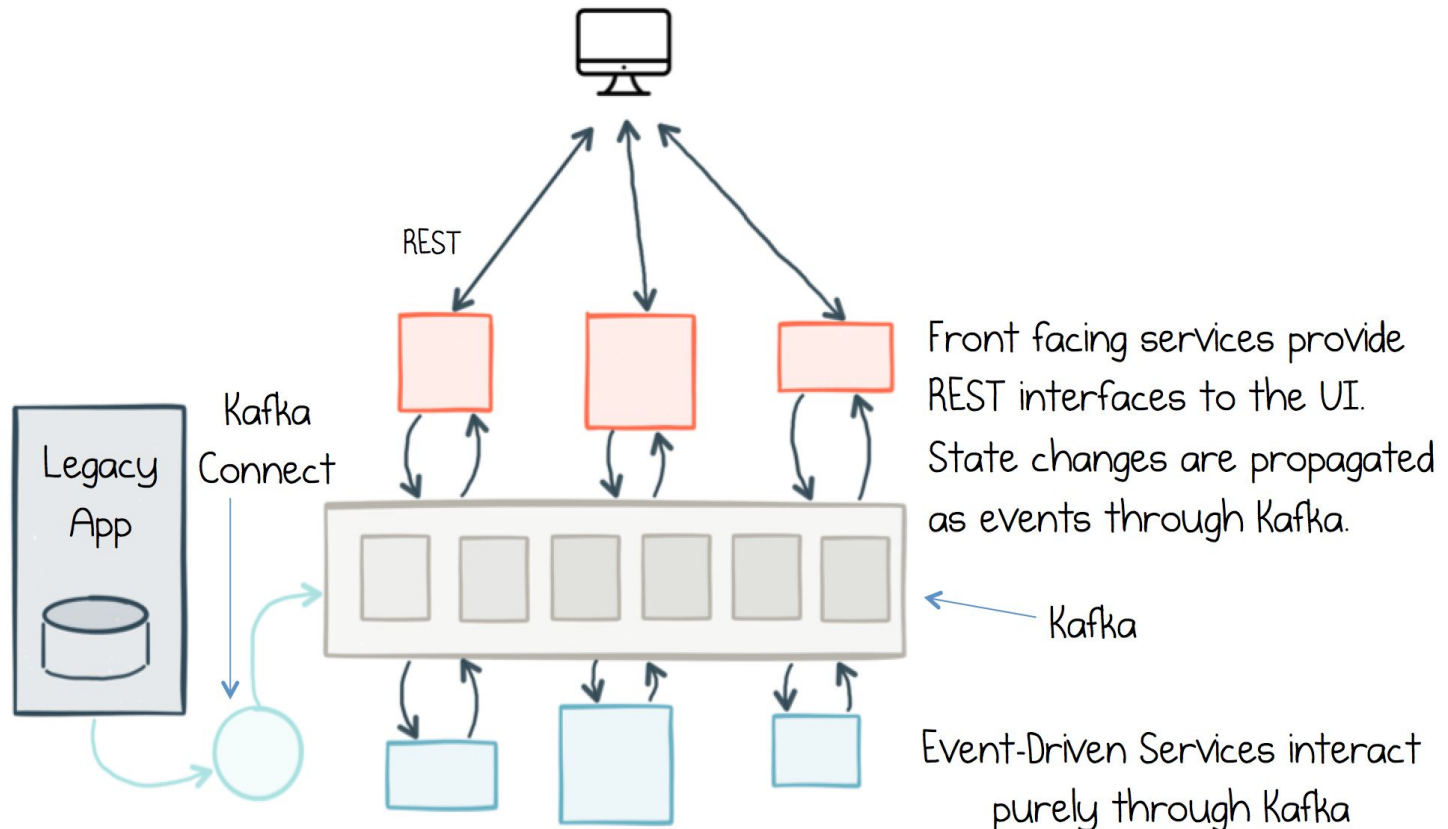
# Web Sockets

Web Sockets es una estrategia comunicacional donde el cliente abre una conexión "permanente" con el servidor y por la misma se realiza una comunicación full duplex entre ambos.



# PubSub via Mensajería


Es una estrategia de comunicación donde el generador de los eventos publica los mensajes en un servicio de mensajería y los interesados en los mismos se registran en el mismo servicio para ser notificados cuando los eventos ocurran.







# Takeaways

- » Existen muchas maneras de definir APIs
  - » Las dos corrientes más grandes son Request/Response y Eventos
  - » Dentro de las mismas hay diversas tecnologías, modos y frameworks que puedo elegir de acuerdo a las necesidades particulares de mis sistema.
  - » Dentro de cada una de ellas muchos de los elementos discutidos cuando vimos gRPC aplican a la hora de pensar/entender la construcción de los mismos.
- 



# CREDITS

Content of the slides:

- » POD - ITBA

Images:

- » POD - ITBA
- » Or obtained from: [commons.wikimedia.org](https://commons.wikimedia.org)

Slides theme credit:

Special thanks to all the people who made and released these awesome resources for free:

- » Presentation template by [SlidesCarnival](#)
  - » Photographs by [Unsplash](#)
- 