Máster en Cloud Apps

Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2019/2020

Trabajo de Fin de Máster

REST, gRPC y GraphQL

Autor: Jesús Ruiz-Ayúcar Vázquez

Tutor: Micael Gallego Carrillo

Contenidos

1. Resumen

El mundo cada día está más conectado: ordenadores, *wearables*, dispositivos móviles, IoT, microservicios… De una forma u otra, todos ellos forman parte de una arquitectura donde los distintos sistemas se comunican entre sí para intercambiar información[1]. A fin de hacer efectiva esta comunicación entre diferentes dispositivos, es necesario recurrir a modelos de transporte basados en la red. Esta comunicación se hace a través de lo que se conoce como API (Interfaz de Programación de Aplicaciones): un *lenguaje* reusable que permite a dos sistemas hablar entre sí a través de la red.

Si bien en algunos casos puede ser recomendable implementar un servicio que se comunique directamente con UDP[2], lo más habitual es utilizar un protocolo *fiable*, como TCP. Además, típicamente no se desea crear un nuevo protocolo de cero, sino que se busca modelar la interfaz de comunicación sobre soluciones aprobadas, flexibles y con alta adopción. Específicamente, dentro del las APIs de petición-respuesta, los estilos más populares son:

- **REST**: un estilo arquitectónico, sin implementación oficial, y basado en recursos, que explota las capacidades del protocolo HTTP y del *hipermedia*.
- **GraphQL**: un ecosistema que provee de un lenguaje de consultas, un IDL (lenguaje de definición de interfaces), y de una implementación oficial.
- **gRPC**: un *framework* RPC (*Remote Procedure Call*) independiente de la plataforma que funciona sobre HTTP/2. Suele hacer uso de Protocol Buffers, un lenguaje de esquemas con una implementación oficial y que permite serializar datos.

Resulta desafiante comparar estas tres soluciones, pues su naturaleza es muy distinta entre sí: mientras REST define un estilo basado por completo en el estándar HTTP, GraphQL es una *implementación* de un lenguaje de consultas, y gRPC es un completo *framework* que oculta la capa de transporte HTTP.

Este proyecto es un esfuerzo por enfrentar estos tres estilos en diferentes categorías: cómo se orientan las primitivas (por recurso o por acción); qué lenguajes de definición de interfaces y de esquemas tienen; de qué manera se realizan las operaciones más comunes (lectura, borrado, búsquedas, paginación, actualizaciones parciales, etc); cómo hacer uso de memorias cachés; qué estrategias de gestión y manejo de errores utilizan; mecanismos de seguridad disponibles; posibilidades de suscripciones a eventos; etc.

Además, se ha creado un repositorio en GitHub[3] a disposición de la comunidad donde se desarrolla en más profundidad cada uno de los conceptos revisados. Dicho repositorio también incluye un proyecto en Node.js en el que se ponen en práctica muchas de las técnicas vistas, y que sirve como material de apoyo para el lector interesado.

---

[1] A esto se le conoce como SOA: Arquitectura Orientada a Servicios.
[2] UDP se usa a menudo en streaming de vídeo, en VoIP o en videojuegos.
[3] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL

2. Introducción y objetivos

Cuando se desea diseñar una API que permita a un servicio habilitar la comunicación con terceros, existe una serie de criterios que se deben valorar a la hora de escoger qué estilo se va a seguir. A continuación, vamos a establecer los principios que nos permitirán categorizar las APIs.

2.1. Paradigmas

Una primera categoría la encontramos a partir de la audiencia a la que esté dirigida. Existen servicios **uno-a-uno**, es decir, con una única posible aplicación cliente. Hay autores que a esto no le consideran una API, sino una *integración*, ya que se espera que una API sea *reusable*. En contraposición, están los servicios **uno-a-muchos**, donde se exponen operaciones genéricas de las que se podría beneficiar un número indeterminado de clientes.

Igualmente, una forma muy común de categorizar APIs es a partir de las operaciones disponibles para cambiar el estado de la aplicación. Encontramos dos tipos: (1) **dirigidas por acciones**, donde existe una lista arbitraria de posibles operaciones, y (2) **dirigidas por recursos**, donde los cambios de estado se producen sobre todo mediante la creación, la eliminación o la actualización de recursos. Por ejemplo, para cancelar el pedido 12:

```
cancelOrder(12) <- Estilo dirigido por acciones
update(12, {cancelled: true}) <- Estilo dirigido por recursos
```

A menudo también se clasifican en función del **modelo de comunicación** utilizado, donde tradicionalmente encontramos dos posibilidades:

- **Sistemas de petición-respuesta**: el servicio envía inmediatamente un mensaje como respuesta a una petición de un cliente. Ejemplos: REST, SOAP, GraphQL, gRPC, Apache Thrift.
- **Sistemas de eventos**: el cliente se suscribe a uno o varios eventos (*topics*), y queda a la espera. Cuando suceda uno de dichos eventos, el servidor mandará un mensaje al cliente. Ejemplos: AMQP, Apache Kafka o WebSockets.

A partir de los paradigmas y de otra serie de restricciones se definen los estilos de API. Básicamente, consisten una serie de *restricciones*[4] con las que se alcanzan unas determinadas características. Por ejemplo, tal y como se desarrolla en el repositorio de este

[4] Introducción a los estilos de APIs, por Zdnek Nemec en la Platform Summit de 2019.

proyecto, REST contiene 6 restricciones[5] (entre ellas, encontramos '*client-server*' o '*stateless*'). Las restricciones, a su vez, conllevan *propiedades*: por ejemplo, si un sistema es *stateless*, entonces entre sus propiedades encontraremos *escalabilidad* y *confiabilidad*.

Decidir qué estilo utilizar vendrá determinado por requisitos del negocio. Un análisis comparativo para la elección del estilo adecuado para un servicio justificaría en sí mismo un estudio pormenorizado que trasciende el objetivo de este proyecto.

2.2. REST, gRPC y GraphQL

Dentro de las APIs de tipo petición-respuesta, encontramos tres grandes bloques muy diferentes entre sí:

- **RPC API** (Llamada a Procedimiento Remoto) - es el más sencillo de todos, donde la API suele estar *dirigida por acciones*. A menudo encontramos implementaciones particulares, como SOAP o JSON-RPC, si bien últimamente se habla más de Apache Thrift o de gRPC.
- **Web API** – cubre las APIs que modelan sus operaciones y recursos alrededor del protocolo web: HTTP. Básicamente, aquí encontramos a REST.
- **Query API** – expresan su estado mediante un grafo de entidades, y exponen una primitivas que permitan realizar consultas a voluntad. Los cambios de estado se realizan por RPC, con lo que se puede entender como un subtipo de RPC API. Tenemos GraphQL.

Se ha elegido un estilo/implementación para cada uno de estos tres bloques:

2.2.1. REST

Acrónimo de *Representational state transfer* (transferencia de estado de representación), consiste en un estilo arquitectónico para sistemas distribuidos de hipermedia. Hablar de Web API es hablar de REST[6].

- Es un estilo arquitectónico que explota HTTP. No hay una implementación estándar.
- Es para sistemas de hipermedia. El motor de la aplicación son los hiperenlaces, que realizan cambios de estados en los recursos.
- Los recursos se manifiestan en diferentes representaciones – texto, JSON o Protocol Buffers. No es necesariamente tipado, pero existen tecnologías que permiten añadir dicho soporte.

Este estilo es tan abierto, que deja del lado del diseñador numerosas decisiones, lo que hace que el costo de desarrollo, la adopción y el *tooling* suponga una importante barrera de entrada. Por contra, promociona enormemente la evolución, la escalabilidad, la confiabilidad y la portabilidad del sistema.

[5] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/rest.md
[6] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/rest.md

2.2.2. gRPC

Desarrollado por Google, gRPC[7] es un *framework* que funciona sobre HTTP/2, lo que le permite ir más allá del modelo tradicional de petición-respuesta ofreciendo *streaming* bidireccional. Típicamente, funcionará con Protocol Buffers.

- Es una implementación, disponible en numerosos entornos, y que oculta por completo HTTP, que es usado como capa de transporte.
- El estado de la aplicación se define a través de llamadas RPC definidas mediante Protobuf. Google recomienda seguir un enfoque *dirigido por recursos*.
- Los recursos se representan mediante Protocol Buffers, luego tiene seguridad de tipos.

Entre las ventajas de gRPC encontramos su gran rendimiento, seguridad de tipos y *streaming* bidireccional. Igualmente, es un sistema muy sencillo. Como contrapartida se encuentra la escalabilidad (*streaming*), capacidad de descubrimiento o el *tooling* disponible.

2.2.3. GraphQL

GraphQL[8] nace a partir de las necesidades de Facebook, y como sustitución de su lenguaje de consultas sobre su anterior API Web: FQL. Se trata de un completo ecosistema en el que destaca un potente lenguaje de consultas que permite a la aplicación cliente seleccionar el conjunto de datos exacto necesario que corresponde con una determinada vista.

- Contiene una implementación oficial[9], y se usa HTTP como capa de transporte.
- Es dirigido por acciones, a través de llamadas RPC que, cuando implican cambios, son llamadas *mutations*. Se definen mediante el GraphQL Schema Language.
- Los recursos se definen con GraphQL Schema Language, y es fuertemente tipado.

Además, incluye herramientas que simplifican el *desarrollo* y la *depuración*, ofrece *suscripciones* mediante WebSockets, y promociona la *introspección* de su interfaz.

Es una alternativa muy recomendable para aplicaciones *frontend*, pues ofrece herramientas para crear con sencillez una API muy flexible y eficiente (reduce los *round-trips*), que además permite el descubrimiento y la seguridad de tipos.

2.3. Objetivos

Hay mucha literatura que, con mayor o menor nivel de detalle, cubre cómo diseñar e implementar una API en cada uno de estos tres estilos. Sin embargo, la mayoría de las veces solo describen las fortalezas de cada uno de ellos, dejando sin cubrir otros casos de uso.

---

[7] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/grpc.md
[8] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/graphql.md
[9] https://github.com/graphql/graphql-js

Por ejemplo, sabemos que GraphQL y gRPC permiten al servidor tener una conexión abierta para enviarle mensajes al cliente. ¿Qué alternativas hay en REST? GraphQL permite seleccionar campos y recursos con un lenguaje de consultas propio. ¿Existe algo parecido en REST o gRPC? REST permite seleccionar representaciones de recursos. ¿Se puede hacer algo parecido en GraphQL o gRPC? ¿Es posible utilizar OAuth2 en gRPC al igual que hacemos en REST o GraphQL? ¿Podemos utilizar actualizaciones parciales en gRPC o GraphQL de manera similar a las que tenemos en REST con una operación PATCH?

El objetivo de este trabajo consiste en someter a prueba cada una de las características y casos de uso de los tres estilos. En particular:

- Explicar cómo definir su interfaz. Esto incluye tanto las operaciones[10] que expone la API, como los esquemas de los datos[11] y el descubrimiento[12] (*discoverability*)
- Mostrar cómo implementar acciones, dividiendo entre:
    - acciones estándar[13], como leer un recurso, seleccionar campos específicos, paginar, buscar, borrar, actualizar o actualizar parcialmente un recurso.
    - acciones personalizadas[14], como acciones de tipo *funciones puras*, mezclar usuarios o realizar una operación por lotes.
- Explorar los mecanismos de seguridad[15] disponibles.
- Analizar las alternativas que permitan crear suscripciones[16].
- Revisar el manejo de errores[17] recomendado.
- Examinar el soporte de memorias cachés[18].

[10] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/contract.md
[11] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/schema_definition.md
[12] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/discoverability.md
[13] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/methods.md
[14] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_custom.md
[15] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/security.md
[16] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/subscriptions.md
[17] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/error_handling.md
[18] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/caching.md

3. Desarrollo

3.1 Contrato

Cada estilo de API tiene un mecanismo diferente para definir sus operaciones disponibles. Se puede encontrar más información en la sección *contract* del repositorio[19].

REST

En REST la interfaz de una API viene definida a partir de los hipervínculos devueltos por un recurso dado, de manera que todo el motor de la aplicación sea la sucesión de dichos enlaces; a esto se le conoce como HATEOAS. Existen numerosos estándares para definir enlaces, como por ejemplo Web Links[20], HAL[21] o JSON API[22]. En este estilo, los recursos se identifican con URIs opacas.

En estilos REST-Like no se suelen usar identificadores opacos, sino URI Templates[23], lo que permite a la aplicaciones cliente invocar recursos a voluntad, como en RPC. En estándares como OpenAPI, se puede además definir formalmente la interfaz mediante `paths`.

GraphQL

Define tres tipos de operaciones:

1. `query` – para exponer un *endpoint* al que realizar consultas.
2. `mutation` – para modificar el estado. Esencialmente, es RPC.
3. `subscription` – permite a una aplicación cliente suscribirse a un cambio por *WebSockets*.

Cada uno se define en su propio *objeto* usando el IDL GraphQL Schema Language:

```
{
  type Query {
      myFirstQuery(param: Int): ReturnType
  }
  schema {
      query: Query
  }
}
```

[19] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/contract.md
[20] https://tools.ietf.org/html/rfc8288
[21] https://tools.ietf.org/html/draft-kelly-json-hal-08
[22] https://jsonapi.org/
[23] https://tools.ietf.org/html/rfc6570

gRPC

En gRPC se utiliza Protocol Buffers como IDL. En particular, se definen mensajes de tipo `service` que, opcionalmente podrán aceptar o devolver *streams*:

```
service Main {
   rpc GetArticles(GetArticlesRequest) returns (stream Articles);
   rpc GetDistance(DistanceRequest) returns (DistanceReply);
}
```

3.2. Esquema

Este apartado cubre cómo describir los datos con que trabajan las operaciones. Se puede encontrar más información en la sección *schema* del repositorio[24].

REST

Determinada metainformación se expresa a través de las cabeceras HTTP, como por ejemplo el idioma (`Content-Language`) o la fecha de modificación (`Last-Modified`).

Especialmente importante es la cabecera `Content-Type`, pues especifica el Media Type[25] del recurso asociado al mensaje HTTP, y donde podemos encontrar desde `text/plain` o `image/png`, hasta formatos extensibles como `application/xml`. En particular, lo más frecuente en REST es encontrar `application/json`.

Es posible también utilizar formatos, como application/schema+json[26] (Usado en OpenAPI) o application/vnd.api+json[27], que permiten describir la estructura de datos de un objeto JSON.

GraphQL

Una de las principales características de este estilo es que se apoya en la representación en modo grafo del estado de la aplicación. El GraphQL Schema Language permite expresar estos grafos principalmente con los siguientes tipos:

- **Escalares**. Enteros, flotantes, cadenas, booleanos o IDs.
- **Enumerados**.
- **Objetos**. Se trata de estructuras compuestas campos. Estos están asociados a un determinado tipo y opcionalmente aceptarán argumentos.
- **Entradas**. Utilizadas para representar argumentos de una operación.

A continuación se muestra un ejemplo de la definición de un objeto

[24] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/schema_definition.md
[25] https://www.iana.org/assignments/media-types/media-types.xhtml
[26] https://tools.ietf.org/html/draft-handrews-json-schema-02
[27] https://jsonapi.org/

```
type Article {
  id: ID!
  title: String!
  date(format: DateFormats = ISO8601): String!
  comments: [Comment!]
}
```

gRPC

En gRPC también podemos describir con precisión los objetos, pero a diferencia de GraphQL, no se pueden añadir argumentos a los campos de cada nodo. Los siguientes tipos son los principales:

- **Escalares**: permite definir precisión y signos, para aprovechar la red.
- **Enumerados**.
- **Mensajes**. Objetos compuestos a partir de otros campos.

```
message Article {
   string id = 1;
   string title = 2;
   string date = 3;
   repeated Comment comments = 4;
}
```

3.3. Get

La más básica de las operaciones, Get puede ser también de las más complejas. Además de una lectura sencilla, también se querrá cubrir cómo seleccionar los campos y los objetos embebidos para evitar *under-fetching* (se recibe menos de lo necesario, y hay que hacer más peticiones) y *over-fetching* (se recibe más de lo necesario, con el coste que conlleva), y cómo seleccionar la representación de un recurso. Se puede encontrar más información en la sección *get* del repositorio[28].

REST

En REST, la operación Get va naturalmente mapeada al método `GET` de HTTP. Y se invoca directamente sobre el identificador del recurso (URI). Es posible seleccionar la representación de un recurso nativamente con HTTP mediante Negociación de Contenido[29].

Si bien de entrada no soporta mecanismos para seleccionar campos específicos, existen técnicas, como Sparse Fieldsets o Embedded resources, que permiten seleccionar campos específicos así como recursos relacionados. Estas técnicas están definidas en

[28] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_get.md
[29] https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation

especificaciones estándar como JSON API[30]. Otras, como OData[31], lo llevan a otro nivel e incluyen un lenguaje similar a SQL que permite definir con precisión la estructura del objeto consultado.

GraphQL

Las consultas en GraphQL se realizan a través de operaciones de tipo `query`. Si bien GraphQL no permite elegir la representación de un recurso (siempre devuelve JSON), sí se pueden añadir argumentos opcionales a un campo para indicar por ejemplo el idioma en el que se quiere leer, la moneda que se desea usar o la unidad de longitud.

La que sin duda es la mayor ventaja de GraphQL es su potente lenguaje de consultas, que permite a una aplicación cliente solicitar con muy sencillez y precisión los campos exactos que necesita.

gRPC

Como con las demás operaciones, en gRPC la operación Get se expresa a través de un rpc. Y aunque funciona sobre HTTP/2 y técnicamente puede transmitir varios formatos, no soporta negociación de contenido.

Con respecto a la selección de campos, al igual que REST, no tiene un mecanismo estándar. Sí existen, sin embargo, patrones comunes, como por ejemplo el uso de FieldMasks[32].

3.4. List

Usada tanto para listar (paginando y ordenador) como para buscar, esta operación presenta también interesantes desafíos. Se puede encontrar más información en la sección *list* del repositorio[33].

REST

Para listar se utiliza una llamada GET a un recurso de tipo colección. Se recomienda expresar la paginación en términos de cursores en lugar de *offsets*, ya que los primeros son opacos y permiten evolucionar la API. Técnicamente la URI debe ser también opaca, pero a menudo, y sobre todo en APIs de estilo REST-Like, se siguen patrones como expresar el límite (máximo de elementos), cursor y criterio de ordenación mediante parámetros GET.

```
GET /univerisities?limit=10&cursor=e73af36376314c7c0022cb1d&sort-by=id
```

[30] https://jsonapi.org/format/#fetching-sparse-fieldsets
[31] https://www.odata.org/getting-started/basic-tutorial/
[32] https://googleapis.dev/nodejs/datacatalog/1.1.0/google.protobuf.html#.FieldMask
[33] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_list.md

Respecto al filtrado para búsquedas, si se trata de un criterio sencillo (nombre = “valor”), se puede describir fácilmente mediante un parámetro GET. Existen soluciones estándar (como en OData[34]) para expresar condiciones y operadores más complejos.

GraphQL

Este estilo da total libertad a la hora de describir cómo listar resultados. Sin embargo, sí existen patrones estándar, recomendadas incluso por la documentación oficial, basadas en el uso de GraphQL Cursor Connections[35]. En este patrón la consulta de lista acepta, o bien el par de argumentos `first` y `after`, o bien el par `last` y `before`. Y la respuesta tiene la siguiente estructura:

```
{
  "pageInfo": {
      "hasNextPage": "<bool>",
      "hasPreviousPage": "<bool>",
      "startCursor": "<cursor opaco al primer elemento>",
      "endCursor": "<cursor opaco al último elemento>"
  },
  "edges": [ elementos ]
}
```

En los nodos tenemos elementos que contienen un cursor (identificador opaco) y un nodo (objeto que contiene).

Dado que las consultas y todos los campos de GraphQL aceptan argumentos opcionales, es trivial añadir condiciones de filtrado a medida para los casos de uso de la API.

gRPC

Al igual que en los demás estilos, se recomienda que la paginación sea basada en cursores, y donde los procedimientos de tipo List reciban un objeto de petición con el tamaño, el cursor y el campo por el que se ordena.

A diferencia de REST y GraphQL, se recomienda utilizar un procedimiento separado para implementar la acción de Search. Este objeto debe aceptar un parámetro de petición indicando todos aquellos criterios de búsqueda aceptados por la API que se desean aplicar.

[34] http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part1-protocol.html#_Toc31358947
[35] https://relay.dev/graphql/connections.htm

3.5 Create

Esta sección cubre la creación de un único documento en una petición. Se puede encontrar más información en la sección *create* del repositorio[36].

REST

Se distinguen dos opciones: colecciones y almacenes.

En las colecciones, el identificador del recurso es elegido por servicio (aunque se puede sugerir un *Slug*[37]). En estos casos, al ser una operación con efectos secundarios y que no es idempotente, se debe utilizar el verbo POST.

En los almacenes, el identificador del recurso se elige por el cliente. Esta operación, al igual que la anterior puede tener efectos secundarios, pero por contra sí es idempotente. Por tanto, se utilizará el verbo PUT.

En cambios casos, si se crea con éxito, el servidor devolverá el nuevo recurso en el cuerpo de la respuesta, junto con una redirección al mismo (cabecera `Location`) y el identificador del recurso cabecera `Content-Location`).

```
POST /articles HTTP/1.1
Content-Type: application/json
Slug: first-article

{'title': 'Post title', 'description': 'Post description'}
```

GraphQL

Si bien técnicamente es posible que una operación `query` tenga *side effects*, se deben utilizar `mutations` para estos casos. Típicamente, se devuelve el nuevo recurso creado.

GraphQL deja mucha libertad al desarrollador a la hora de cómo implementar la creación de un elemento:

```
type Mutation {
  createArticle(article:ArticleInput): Article
}
```

Sin embargo, se recomienda usar operaciones idempotentes[38], donde cada petición tiene un identificador que impediría crear más de una vez el recurso.

[36] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_create.md
[37] https://tools.ietf.org/html/rfc5023#section-9.7
[38] https://shopify.engineering/building-resilient-graphql-apis-using-idempotency

gRPC

Siguiendo las recomendaciones de diseño, se utilizará un rpc específico para la creación.

```
rpc CreateArticle(CreateArticleRequest) returns (Article);
```

De nuevo, se considera una buena práctica ofrecer un identificador único asignado por el cliente, y dejar que el servidor decida cómo proceder (devolver un código de error estándar, `ALREADY_EXISTS`, o crear el recurso asignándole un nuevo identificador). En caso de éxito, se devuelve el nuevo recurso creado.

3.6. Update

Consiste en actualizar un recurso, y típicamente se encuentran dos enfoques: reemplazo total o actualización parcial[39]. Además, se deben considerar cómo proceder con los campos de solo lectura (por ejemplo, fecha de creación de un recurso). Se puede encontrar más información en la sección *update* del repositorio[40].

REST

Un reemplazo de objeto es una operación idempotente y con efectos colaterales. Se usará PUT, pues es el método que encaja con la semántica de la operación. El cuerpo de la petición contendrá el recurso. En caso de éxito, se devolverá el recurso actualizado.

Si se proporcionan campos de solo lectura con valores inválidos, se puede o bien ignorarlos o bien devolver un código de error `409 - Conflict`.

Las actualizaciones parciales se realizan con el método PATCH. Existen varios estándares para expresar estas actualización, como JSON Merge Patch[41] y JSON Patch[42].

En los dos casos se recomienda utilizar peticiones condicionales, bien por id (`ETag`) o por fecha (`Last-Modified`) para que el servidor proteja ante condiciones de carrera.

GraphQL

Esta operación se expresa mediante una *mutation*. Por ejemplo:

```
type Mutation {
  updateProduct(id: ID!, product:ProductInput!): Product
}
```

[39] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_update_partial.md
[40] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_update.md
[41] https://tools.ietf.org/html/rfc7396
[42] https://tools.ietf.org/html/rfc6902

GraphQL no maneja actualizaciones parciales[43], pero da suficiente libertad para permitir crear soluciones de compromiso. Un patrón muy común consiste en definir una operación que acepte todos los campos y que sean opcionalmente indefinidos, de manera que solo los campos definidos se actualizarán:

```
type Mutation {
  patchProduct(id: ID!, name:String!, price:Int!): Product
}
```

gRPC

De manera análoga a la creación, para la actualización se creará un nuevo rpc:

```
rpc UpdateArticle(UpdateArticleRequest) returns (Article);
```

Si bien esto es válido, se recomienda dentro de lo posible utilizar un campo de tipo `FieldMask`[44], con lo que se lograría poder expresar actualizaciones parciales equivalentes a las de una operación PATCH de REST.

3.7. Delete

La operación Delete representa el borrado de un único recurso. Sin considerar validaciones, la mayor controversia está en cuanto al mensaje devuelto como respuesta, y a si considerar esta operación como idempotente. Se puede encontrar más información en la sección *delete* del repositorio[45].

REST

Existe un método específico en HTTP para el borrado de un recurso: `DELETE`. De este modo, para borrar un recurso, tan solo se invocará al método `DELETE` sobre este y, en caso de éxito, se devolverá un código de respuesta "`204 - No content`".

Esta operación no está libre de controversia, pues según HTTP se define como idempotente (dos llamadas al mismo deben producir el mismo resultado). Se plantea entonces la duda de qué se debe hacer al intentar borrar de segundas un recurso que ya se borró: devolver un mensaje de error porque el recurso no existe o mantener una lista de recursos borrados.

Algunos autores[46] consideran que el efecto debe ser idempotente, pero no necesariamente la respuesta, con lo que es aceptable recibir una segunda petición de borrado y responder con un mensaje de error porque este no existe.

GraphQL

[43] https://medium.com/@__xuorig__/graphql-mutation-design-batch-updates-ca2452f92833
[44] https://cloud.google.com/apis/design/standard_methods#update
[45] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_delete.md
[46] https://cloud.google.com/apis/design/standard_methods#delete

Al igual que con todas las operaciones con efectos colaterales, GraphQL utilizará una `mutation`.

```
deleteArticle(id: ID!): Article
```

Sin embargo, no hay consenso en qué tipo de objeto se debe devolver. Por ejemplo, al eliminar un proyecto, GitHub devuelve una representación del dueño del mismo, lo cual puede ser apropiado para su caso de uso. Otras APIs, en cambio, devuelven el recurso que se acaba de eliminar.

gRPC

En gRPC se creará un rpc específico para borrar recurso:

```
rpc DeleteArticle(DeleteArticleRequest) returns (Empty);
```

Como respuesta, es posible tanto devolver un objeto (ya sea el recurso recién borrado, o algún otro recurso relacionado), o una respuesta vacía. Sin embargo, se recomienda replicar el funcionamiento de HTTP y por tanto de REST, y devolver una respuesta vacía.

3.8. Otras acciones

Si bien es técnicamente posible expresar toda una API en términos de las operaciones Get, List, Create, Update y Delete, a menudo se desea mejorar la experiencia de desarrollo ofreciendo nuevas primitivas. Esto puede cubrir desde solicitar una operación similar a una *función pura* (cuya entrada se determina a partir de la entrada), como por ejemplo convertir de Fahrenheit a Celsius, hasta operaciones de cambio de estado, mezcla de dos recursos en uno, u operaciones por lotes. Se puede encontrar más información en la sección *custom methods* del repositorio[47].

REST

Una operación sin efectos colaterales se puede expresar mediante GET. Por ejemplo, para calcular la distancia entre dos ciudades.

Para realizar un cambio de estado existen varios enfoques:

1. En REST, el recurso expondrá los cambios de estado posibles mediante hipervínculos para, por ejemplo, cancelar un recurso de tipo pedido.
2. Otra opción pasa por permitir actualizar un campo que define el estado.
3. A veces es también posible extraer un recurso que represente el nuevo estado.

En APIs REST-Like pueden usarse también recursos de tipo controller. Este tipo de recursos son básicamente RPCs a acciones específicas, por ejemplo, copiar o mover un recurso, mezclar dos recursos u operaciones en lote.

[47] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/method_custom.md

GraphQL

Las acciones sin efectos colaterales se describirán mediante nuevas `queries`.

Los cambios de estado se realizarán mediante `mutations`, que en esencia son llamadas RPC. Es posible expresarlas siguiendo un enfoque dirigido por recursos (con actualización total, o parcial), o dirigido por acciones. Por ejemplo:

```
type Mutation {
  updateMicrowave(microwave: Microwave!): Microwave!
  updateMicrowaveStatus(microwaveId: ID!, status: Status!): Microwave!
  turnMicrowaveOn(microwaveId: ID!): Microwave!
}
```

Cualquier otra operación se podrá expresar de manera natural en una nueva `mutation`. GraphQL, además, permite que una `query` o una `mutation` contengan muchas acciones. Esto abre la puerta a operaciones por lotes.

gRPC

Al igual que en REST y GraphQL, para una operación sin *side effects* se puede usar una acción Get.

Con respecto a cambios de estado, se recomienda seguir un patrón dirigido por recursos, con lo que se abre la puerta a utilizar un campo o recurso que represente el cambio de estado, como ocurría en REST, y combinarlo con una actualización total o parcial del recurso principal. Finalmente, cualquier otra operación se puede expresar a través de un rpc.

3.9. Manejo de errores

En el contexto de una petición puede surgir un error, ya sea porque la petición es inválida (*el recurso no existe*), o porque ocurrió un error interno (*no hay espacio en disco*). Las APIs deben poder devolver de manera expresiva los errores al cliente. Se puede encontrar más información en la [sección *error handling* del repositorio](https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/error_handling.md)[48].

REST

Se deben utilizar los [mensajes de error de HTTP](https://tools.ietf.org/html/rfc7231#section-6)[49] siempre que sea posible. Por ejemplo, si se solicita un recurso que no existe, se devolverá un código de estado 404. Esto, sin embargo, a menudo puede resultar confuso. Supongamos que se solicita el recurso '`/departments/51/employees?id=Smith`'. A priori, puede significar tanto que el departamento 51 no existe, como que dicho departamento no tiene ningún empleado Smith.

[48] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/error_handling.md
[49] https://tools.ietf.org/html/rfc7231#section-6

Para poder expresar los errores con más detalle, se recomienda que la API REST utilice el *payload*. Existen varios estándares para esto, como JSend[50] o como la RFC 7807[51]. Por ejemplo, un error en RFC 7807:

```
{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "balance": 30,
  "accounts": ["/account/12345", "/account/67890"]
}
```

GraphQL

Las respuestas de GraphQL son un objeto que puede tener dos campos: `data` y `errors`. El primero contiene el resultado de una petición satisfactoria, mientras que el segundo devuelve un array de errores, en caso de que los haya. La mayoría de implementaciones de servidores GraphQL completarán automáticamente los errores a partir de las excepciones no capturadas. Algunos, como Apollo Server, incluyen una lista predeterminada de errores[52].

Además, una aplicación GraphQL puede también verse afectada por errores HTTP, por ejemplo por problemas en la red. Este tipo de errores se pueden manejar con normalidad como en una API REST, si bien la mayoría de clientes GraphQL abstraen de los mismos.

gRPC

En el caso de gRPC, se esconde HTTP, por tanto no se pueden usar sus códigos de estado. En su lugar, se define una lista estándar de códigos de error[53], independiente de Protocol Buffers, donde se encuentran mensajes como `GRPC_STATUS_UNIMPLEMENTED` o `GRPC_STATUS_INTERNAL`.

Esta lista, sin embargo, es limitada y puede no ser suficiente para expresar adecuadamente un mensaje de error. Para esto, se recomienda utilizar Modelo de Errores de Google[54]. Este modelo sugiere que las respuestas pueden ser o bien del tipo esperado, en caso de éxito, o bien de tipo `Status`, en caso de error. Este mensaje Status contendrá detalles como un código de error[55], una descripción, y una lista de detalles.

---

[50] https://github.com/omniti-labs/jsend
[51] https://tools.ietf.org/html/rfc7807
[52] https://www.apollographql.com/docs/apollo-server/data/errors/
[53] https://grpc.io/docs/guides/error/#error-status-codes
[54] https://cloud.google.com/apis/design/errors
[55] https://github.com/googleapis/googleapis/blob/master/google/rpc/code.proto

3.10. Seguridad

En esta sección, nos centramos en la **autenticación** (identificar quién hace una petición) **autorización** (comprobar si se tienen permisos para realizar la operación), **integridad** (los datos se representan correctamente) y **confidencialidad** (los datos están protegidos de accesos de terceros). Se puede encontrar más información en la sección *security* del repositorio[56].

REST y GraphQL

Si bien GraphQL funciona sobre HTTP, no lo oculta por completo. Esto hace que la mayoría de técnicas de seguridad de REST sean igualmente aplicables a GraphQL.

Para poder proporcionar tanto integridad como confidencialidad, se utiliza *Transport Layer Security* (TLS)[57].

Si bien se pueden usar *cookies* para proporcionar tanto autenticación como autorización, lo habitual es apoyarse en la cabecera `Authorization`, usando alguno de los tipos disponibles en IANA[58]:

- **Basic** - Se envía el nombre de usuario y la contraseña. Debe funcionar sobre TLS.
- **Digest** - El servidor envía un desafío con un nonce, de manera que el cliente no envía la contraseña, sino un token generado a partir del desafío. Impide *replay attacks*.
- **Bearer** - Cualquier petición en posesión de un determinado token, podrá usarlo para acceder a los recursos autorizados. Solo debe funcionar sobre TLS. Típicamente se genera como resultado de una negociación OAuth2. Es el mecanismo más común. A menudo se utiliza en conjunto con **JWT**.

Otro mecanismo muy popular consiste en **API Keys**. Están presentes desde los comienzos de las APIs Web, y consisten en un token generado por el administrador de la API y permite identificar o bien a una aplicación/organización que actúa en su propio nombre, o bien a una aplicación que actúa en nombre de un usuario. Se pueden transportar por casi cualquier canal: parámetros GET, cookies, Authorization de tipo Basic o de tipo Bearer…

Además, los navegadores web por seguridad no pueden acceder a recursos de otros dominios. A veces se desean relajar estas medidas para permitir que terceros puedan acceder a nuestros recursos. En este caso, es posible utilizar CORS[59].

gRPC

En el caso de gRPC, para ofrecer integridad y confidencialidad, también se utiliza TLS (de facto incluida siempre en HTTP/2 a través de la extensión ALPN).

Con respecto a la autenticación, por defecto soporta tres mecanismos:

- TLS, a través de certificados.
- ALTS, similar a TLS pero específico para Protocol Buffers en Google Cloud Platform.
- Token-based for Google, mecanismo basado en tokens solo para servicios Google.

---

[56] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/security.md
[57] https://tools.ietf.org/html/rfc8446
[58] https://www.iana.org/assignments/http-authschemes/http-authschemes.xhtml
[59] https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS

gRPC secuestra la sesión HTTP, con lo que no es trivial leer y escribir sus cabeceras. Sin embargo, a menudo se hacen soluciones utilizando la metadata (lista de clave-valor que viaja en una conexión gRPC) como medio para enviar tokens, por ejemplo OAuth.

3.11. Suscripciones

Este proyecto cubre estilos de APIs dirigidas por un modelo de comunicación de petición-respuesta. Este modelo de comunicación puede ser un derroche de recursos. Veamos posibles mejoras o alternativas en cada uno de los estilos. Se puede encontrar más información en la sección *subscriptions* del repositorio[60].

REST

Un servicio REST, por definición, es sin estado, y por tanto no puede mantener una conexión abierta con el cliente.

Lo que sí es posible es recurrir a **WebHooks**: un mecanismo que permite a una aplicación suscribir una *callback* URL para que sea invocada cada vez que un determinado evento suceda. Esta suscripción puede ser global (todos los eventos de cambio posibles), por un tema particular, o parametrizado (solo ciertas notificaciones de un tema particular). Naturalmente, este modelo solo funciona cuando la aplicación cliente dispone de servidor web.

Existen también otras alternativas que habilitan la suscripción también a navegadores:

- **Transfer-Encoding: Chunked**. En este caso, se obtiene un efecto similar al streaming de HTTP/2, manteniendo una conexión TCP abierta entre el servidor y el cliente.
- **Long polling**. Se mantiene una conexión abierta indefinidamente hasta recibir la respuesta. Una vez recibida, se abre una nueva conexión.

Estas dos tecnologías han quedado obsoletas tras la aparición de WebSockets y *streaming* de HTTP/2.

GraphQL

A diferencia de REST, GraphQL sí maneja el concepto de suscripción. Además de las operaciones de tipo `query` y `mutation`, el estándar define un tercer tipo, llamado `subscription`, que permite crear suscripciones opcionalmente parametrizadas (lo cual permite seleccionar eventos específicos dentro de un tema en particular). Internamente, se utiliza una conexión a través del protocolo WebSocket. Al igual que en el resto de operaciones de GraphQL, se puede elegir la *forma* de la respuesta que se desea recibir.

gRPC

Uno de los mayores avances de gRPC respecto a otros estilos que operan sobre HTTP es que explota por completo y simplifica las posibilidades de streaming bidireccional nativas de HTTP/2.

[60] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/subscriptions.md

Cada conexión abierta (*channel*, en terminología gRPC) entre un cliente un servidor gRPC, puede exponer uno o más *endpoints* RPC. Cada uno de estos puede definir su modo:

- **Unario**: modelo tradicional de petición-respuesta.
- **Streaming de servidor**: el cliente recibe un flujo de datos del servidor.
- **Streaming de cliente**: el cliente envía un flujo de datos al servidor. Típicamente, este último envía su respuesta cuando el *streaming* ha finalizado.
- **Streaming bidireccional**: se crean dos flujos independientes, uno desde el cliente y otro desde el servidor.

3.12. Cacheo

El cacheo permite mejorar el rendimiento de una aplicación, reduciendo el número de acceso a los recursos. En primer lugar, vamos a diferenciar entre varios tipos de caché:

- **De aplicación**. En el servidor, usando por ejemplo Redis. Independiente de la API.
- **Web**. Se sitúa entre el cliente y el servidor.
- **Local**. Ocurre en el cliente. Los identificadores únicos simplifican su implementación.

Se puede encontrar más información en la sección *caching* del repositorio[61]. Vamos a centrarnos en los dos últimos.

REST

Gracias al diseño de HTTP, orientado a recursos y con identificadores únicos, REST puede beneficiarse de varios mecanismos disponibles, tanto para cachés web como locales.

Además, HTTP permite de manera nativa gestionar la caché de un recurso mediante la cabecera `Cache-Control`. Por ejemplo, se puede impedir que un valor sea almacenado usando el valor `no-store`, o que solo pueda ser almacenado en la aplicación cliente (son datos privados) usando el valor `private`. Para gestionar el tiempo durante el cual se puede considerar que un recurso está fresco, se puede utilizar la propiedad `max-age`.

```
Cache-Control: public, max-age=604800
```

Pasado dicho tiempo, se recargaría el recurso en caso de ser solicitado de nuevo. Si este hubiera incluido una cabecera `ETag` o `Last-Modified`, se podría revalidar.

Por norma general, las aplicaciones limitarán sus cachés solo a las peticiones que utilizan el método GET.

Es igualmente importante señalar que las aplicaciones con peticiones personalizables (por ejemplo, con búsquedas avanzadas, o que hacen uso de *sparse fieldsets*) podrán beneficiarse en menor medida de estas cachés.

[61] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/caching.md

GraphQL

Si bien funciona sobre HTTP, los servicios de GraphQL solamente exponen un *endpoint* que, además, suele usarse a través del método POST. Sin embargo, al ser usado sobre GET sí es susceptible de beneficiarse del cacheo de HTTP.

Además, se recomienda tener identificadores de objetos globales[62] para facilitar el almacenamiento de cachés en local.

gRPC

Si bien, al estar basado en HTTP/2 hace un uso más eficiente de los recursos de red, en gRPC no es sencillo beneficiarse de la caché de HTTP. Sí permite establecer identificadores ETag[63].

Al igual que en GraphQL, es posible también seguir una estrategia que permita crear identificadores únicos de recursos y que, por tanto, facilite la creación de una caché local.

3.13. Descubrimiento

Descubrimiento, del inglés *discoverability*, es la capacidad de que una API exponga la interfaz con la que se puede actuar sobre sus recursos. Se puede encontrar más información en la sección *discoverability* del repositorio[64].

REST

En una API REST los hipervínculos funcionan como motor de la aplicación; es decir, cada recurso contiene una lista con las operaciones que tiene asociada, lo cual es una forma de ofrecer *discoverability*. El grado de información que se da en los hipervínculos dependerá del estándar que se haya seguido; algunos permiten incluso definir el esquema de los argumentos que recibe una acción.

Además, es posible hacer uso del método HTTP `OPTIONS`. Este devuelve una lista de los métodos HTTP disponibles sobre un determinado recurso.

En servicios de tipo REST-Like, es decir, que no siguen la restricción HATEOAS, es posible describir la interfaz de la aplicación en un lenguaje formal. Un ejemplo es OpenAPI, que permite desarrollar un ecosistema alrededor que permita, por ejemplo, autogenerar una librería para el cliente.

GraphQL

Una de las características nativas de GraphQL es el soporte a la introspección[65]. Por defecto, las implementaciones incluyen una serie de campos de búsqueda para consultar la estructura del servicio.

---

[62] https://graphql.org/learn/global-object-identification/
[63] https://cloud.google.com/apis/design/design_patterns#etags
[64] https://github.com/MasterCloudApps-Projects/REST-gRPC-GraphQL/blob/master/docs/usage/discoverability.md
[65] https://graphql.org/learn/introspection/

Por ejemplo, la siguiente `query` permite obtener el nombre de cada tipo de dato de un servicio GraphQL:

```
{
  __schema {
      types {
          name
      }
   }
}
```

Igualmente, existe interés en utilizar hipervínculos en las APIs GraphQL[66].

gRPC

Al igual que ocurre con OpenAPI, gRPC permite definir la interfaz de un servicio mediante un lenguaje formal: Protocol Buffers. A partir de esta definición, es posible (aunque no es necesario) autogenerar el código de la librería cliente: usando la herramienta `protoc`.

No tiene soporte nativo de introspección, pero existen iniciativas avanzadas para ofrecer reflexión de un servidor gRPC[67].

[66] https://medium.com/@__xuorig__/graphql-mutation-design-hypermedia-graphql-api-faf03f3a898a
[67] https://github.com/grpc/grpc/blob/master/doc/server-reflection.md

4. Conclusiones

Como se explicaba en la Introducción, existe mucha literatura que profundiza en cada estilo por separado. Durante la preparación de este trabajo, también he tenido la oportunidad de encontrar muchas publicaciones y conferencias donde se contrastan, a muy alto nivel, las diferencias entre diferentes estilos. Sin embargo, no es fácil encontrar publicaciones donde se baje al detalle sobre cómo implementar casos de uso particulares en cada uno de ellos. Este trabajo nacía con ese claro objetivo: enfrentar las implementaciones de APIs en tres de los estilos más populares a día de hoy, tanto desde un marco más teórico, como aterrizando los conceptos en un proyecto real. Si bien han quedado puntos por explorar, considero que sí se ha logrado con éxito describir las principales operaciones y decisiones de diseño.

Hemos visto que los tres forman parte de ecosistemas muy maduros en los que, de una forma u otra, existen mecanismos para poder sortear sus limitaciones iniciales. Por ejemplo:

- Existen maneras de añadir validación de tipos, auto-generación de código o lenguajes de consultas a REST.
- Podemos tener arquitecturas RPC dirigidas por recursos.
- Existen modelos que permiten enriquecer la gestión de errores de GraphQL.

Sin embargo, existen barreras tanto culturales como tecnológicas en cada uno de ellos:

- **REST** es muy potente y sobre todo flexible, pero es también complejo, y requiere que los desarrolladores, tanto del lado del servidor como del cliente, dominen HTTP y los diferentes estándares sobre los que se apoye la implementación final.
- **gRPC** es un sistema muy eficiente, pero probarlo no es tan sencillo como usar cURL, un navegador o Postman. Aún no se ha popularizado su uso en Web. Es difícil explotar la caché de HTTP.
- **GraphQL** es muy prometedor, pero al hacer *tunnelling* sobre HTTP, no puede aprovechar su caché, y dificulta casos de uso que no hemos cubierto, como por ejemplo subida de ficheros, o gestión (rendimiento) de consultas complejas.

Pero sin duda, algo de lo que más me ha llamado la atención ha sido toda la ambigüedad alrededor de REST. Tanto GraphQL como gRPC, al tener especificaciones, tienen un alcance mucho más definido. Sin embargo, prácticamente todos los contenidos acerca de REST hablan sobre estilos que, con suerte, alcanzan el nivel 2 del Maturity Level de Richardson: verbos HTTP. Es realmente complejo encontrar clientes reales que funcionen siguiendo hiperenlaces. Si además nos centramos en encontrar APIs consumidas de manera desatendida (por ejemplo, por servicios que consumen otras APIs), en contraposición a aquellas controladas por un usuario final, entonces ya la tarea se torna casi imposible.

En cualquier caso, opino que una Web API tipo RPC como OpenAPI, pese a no ser REST puro, no desmerece en absoluto su adopción (de hecho es un enfoque muy similar a gRPC).

5. Trabajos futuros

Otra de las mayores conclusiones a las que he llegado es lo grande que es el universo de las APIs. Existen, no solo libros y ensayos que las analizan desde todos los ángulos imaginables, sino empresas, comunidades de usuarios o conferencias dedicadas en exclusiva al estudio, promoción y explotación de las APIs. Este trabajo no es sino un pieza más, y al que se le puede dar continuidad desde muchos frentes:

Como se ha visto, REST es una arquitectura muy compleja y no se ha llegado a explorar en detalle. Sería interesante profundizar más allá de un sencillo ejemplo académico en el **diseño e implementación de una API RESTful**, utilizando alguno de los muchos estándares de hiperenlaces disponibles, y que realmente estos sirvan como máquina de estados. Igualmente interesante sería enfrentarla con una API REST-Like que haga uso de alguna de las especificaciones más populares, como **OpenAPI Specification** o **RAML**.

El objetivo de crear un pequeño servicio de ejemplo en Node.js y Express fue poder enseñar una implementación real que ponga en práctica los conceptos vistos. En este sentido, sería interesante continuar este trabajo añadiendo una **implementación en algún otro lenguaje** y *framework*, como por ejemplo Java junto con Spring Boot o PHP con Symfony.

Por otro lado, y si bien se ha usado o detallado cómo trabajar con WebSockets (en GraphQL), con *streams* HTTP/2 (en gRPC) o con WebHooks (en REST), no se ha llegado a profundizar en el *diseño* de **APIs dirigidas por eventos**. En particular, Apache Kafka está ganando mucha tracción recientemente, y abre la puerta a otras herramientas y tecnologías vistas durante el Máster, como por ejemplo CQRS y *Event Sourcing*.

Sabemos que REST, al basarse en HATEOAS, ofrece una arquitectura muy resiliente; que GraphQL permite conocer en detalle qué campos se usan y por quién; y que Protocol Buffers simplifica la modificación de sus esquemas. Con todo, la **evolución y el versionado de APIs** es otro asunto que ha quedado por explorar.

Al igual que otros muchos *topics*:

- Implementación de OAuth, incluyendo *scope*.
- Cultura DevOps: Ley de Conway, *lead time*, despliegues, A/B testing.
- Testing de APIs.
- API como producto: API Management[68].
- Rendimiento: multiplexación en HTTP/2[69].

[68] https://www.redhat.com/es/topics/api/what-is-api-management
[69] https://nordicapis.com/does-http-multiplexing-make-graphql-obsolete/

6. Bibliografía

Brenda Jin, Saurabh Sahni, Amir Shevat. “Designing Web APIs”. O'Reilly Media, Inc., 2018

Subbu Allamaraju. “RESTful Web Services Cookbook”. O'Reilly Media, Inc. Marzo de 2010

Mark Masse. “REST API Design Rulebook”. O'Reilly Media, Inc. Octubre de 2011

Google Cloud. “API Design Guide”. Última modificación: Julio de 2020. https://cloud.google.com/apis/design

Zdenek “Z” Nemec. “What API: Your Guide to API Styles”. Noviembre de 2019 https://youtu.be/gRZbgsmDj_0

Phil Sturgeon. “Understanding RPC, REST and GraphQL”. Enero de 2018. https://apisyouwonthate.com/blog/understanding-rpc-rest-and-graphql

Phil Sturgeon. “Picking the right API Paradigm”. Mayo de 2018. https://apisyouwonthate.com/blog/picking-the-right-api-paradigm

Thomas Bush. “An Expert’s Guide to Choosing the Right API Style”, Abril de 2020. https://nordicapis.com/an-experts-guide-to-choosing-the-right-api-style/

Phil Sturgeon. “GraphQL vs REST: Overview”. Enero de 2017. https://phil.tech/2017/graphql-vs-rest-overview/

Zdenek "Z" Nemec. “REST vs. GraphQL: A Critical Review”. Abril de 2019. https://goodapi.co/blog/rest-vs-graphql

Martin Nally. “API design: Understanding gRPC, OpenAPI and REST and when to use them”. Abril de 2020. https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them

Thomas Betts, Charles Humble. “Software Architecture and Design InfoQ Trends Report”. Abril de 2020. https://www.infoq.com/articles/architecture-trends-2020/

Roy Thomas Fielding, Tesis Doctoral “Architectural Styles and the Design of Network-based Software Architectures”. Universidad de California, Irvine. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Andreas Reiser. “Why HATEOAS is useless and what that means for REST”. Febrero de 2018. https://medium.com/@andreasreiser94/why-hateoas-is-useless-and-what-that-means-for-rest-a65194471bc8

Pakal de Bonchamp. “REST is the new SOAP”. Diciembre de 2017. https://www.freecodecamp.org/news/rest-is-the-new-soap-97ff6c09896d/

Google Cloud. Apigee. “Web API Design: The Missing Link”. 2018. https://cloud.google.com/files/apigee/apigee-web-api-design-the-missing-link-ebook.pdf

Phil Sturgeon. “You Might Not Need GraphQL”. Abril de 2017. https://blog.runscope.com/posts/you-might-not-need-graphql

Jean de Klerk. “HTTP/2: Smarter at scale”. Julio de 2018. https://www.cncf.io/blog/2018/07/03/http-2-smarter-at-scale/

Roy Thomas Fielding. “REST APIs must be hypertext-driven”. Octubre de 2018.
https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Asbjørn Ulsberg. “The REST And Then Some”. Octubre de 2016.
https://youtu.be/QIv9YR1bMwY

Asbjørn Ulsberg. “API Change Strategy”. Marzo de 2018.
https://nordicapis.com/api-change-strategy/

Vinay Sahni. “Best Practices for Designing a Pragmatic RESTful API”. Mayo de 2020.
https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

Tom Huston. “What Is Hypermedia?”. Noviembre de 2018.
https://smartbear.com/learn/api-design/what-is-hypermedia/

Brian Mulloy. “HATEOAS 101: Opinionated Introduction to a REST API Style”. 2012.
https://youtu.be/6UXc71O7htc

Duncan Cragg. “Minted Media Types are Usually Less RESTful Than JSON”. Mayo de 2011.
http://duncan-cragg.org/blog/post/minting-media-types-usually-less-restful-using-raw/

Mozilla Foundation. “Content negotiation”. Última modificación: Noviembre de 2020.
https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation

Todd Jefferson. “Building Resilient GraphQL APIs Using Idempotency”. Agosto de 2019.
https://shopify.engineering/building-resilient-graphql-apis-using-idempotency

Arnaud Bezançon. “GraphQL mutations: Partial updates implementation”. Mayo de 2017.
https://medium.com/workflowgen/graphql-mutations-partial-updates-implementation-bff586bda989

Marc-André Giroux. “GraphQL Mutation Design: Batch Updates”. Marzo de 2018.
https://medium.com/@__xuorig__/graphql-mutation-design-batch-updates-ca2452f92833

Phil Sturgeon. “Representing State in REST and GraphQL”. Agosto de 2017.
https://apisyouwonthate.com/blog/representing-state-in-rest-and-graphql

Konstantin Tarkus. “Custom errors and error reporting in GraphQL”. Marzo de 2016.
https://medium.com/@koistya/validation-and-user-errors-in-graphql-mutations-39ca79cd00bf

Konstantin Tarkus. “Validation and User Errors in GraphQL Mutations”. Marzo de 2016.
https://medium.com/@koistya/validation-and-user-errors-in-graphql-mutations-39ca79cd00bf

Mozilla Foundation. “HTTP authentication”. Última actualización: Julio de 2020.
https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

Mozilla Foundation. “Cross-Origin Resource Sharing (CORS)”. Última actualización: Noviembre de 2020. https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Guy Levin. “RESTful API Authentication Basics”. Noviembre de 2016.
https://blog.restcase.com/restful-api-authentication-basics/

OWASP. “REST Security Cheat Sheet”.
https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

Mozilla Foundation. “Transfer-Encoding”. Última actualización: Noviembre de 2019. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding

Mark Nottingham. “Caching Tutorial”. 2013. https://www.mnot.net/cache_docs/

Jeff Posnick, Ilya Grigorik. “Prevent unnecessary network requests with the HTTP Cache”. Abril de 2020. https://web.dev/http-cache/

Eugen Paraschiv. “REST API Discoverability and HATEOAS”. Julio de 2020. https://www.baeldung.com/restful-web-service-discoverability

Anexo I - Cuadro resumen

Cuadro resumen con los detalles de implementación de cada uso:

Uso	**REST**	**GraphQL**	**gRPC**
Contrato	HATEOAS o OpenAPI	GraphQL Schema Language: operaciones	Protocol Buffers: `rpc`
Esquema	Cabeceras. `Media Type`. JSON Schema	GraphQL Schema Language	Protocol Buffers: messages
Acciones estándar	`GET`, `POST`, `PUT`, `PATCH`, `DELETE`	`query` y `mutation`	Operaciones `rpc`
Get	`GET`	`query`	Operacion `Get`
Get (representación)	`Content Negotiation`	Solo JSON	Solo uno. Protobuff
Get (a medida)	*Sparse fieldsets*. Recursos embebidos.	Soporte nativo.	`FieldMask`
List	`GET`. Paginación, ordenado y filtrado a medida.	`query`. Paginación y ordenador estándar.	Operaciones `List` y `Search`.
Create	`POST` o `PUT`	`mutation`	Operacion `Create`
Update	`PUT`	`mutation`	Operación `Update`
Update (parcial)	`PATCH`	No soportado	Operación `Update` con `FieldMask`.
Delete	`DELETE`	`mutation`	Operación `Delete`
Acciones a medida	HATEOAS, `GET` o `POST`	`query` o `mutation`	`rpc`
Manejo de errores	HTTP. Extensible.	Propiedad `errors`	Lista Estándar. Google Error Model.
Seguridad	`Bearer`, OAuth, API Keys, TLS, CORS	`Bearer`, OAuth, API Keys, TLS, CORS	TLS, ALTS, token Google), a medida
Subscripciones	No soportado. WebHook y HTTP *streaming*	`subscription`	HTTP/2 *streaming*
Caché	Local y HTTP	Local y HTTP (parcial)	Local y HTTP (parcial)
Descubrimiento	HATEOAS, `OPTIONS` o OpenAPI	Introspección nativa	No soportado. Reflexión.