Kauã Rieper Ribeiro

Estilos de API e trade-offs: explique como REST, GraphQL e gRPC atendem casos distintos. Discuta latência, acoplamento, versionamento, cache e granularidade. Mostre quando não usar cada um.

**REST:** No REST você não escolhe cada campo, você escolhe o recurso que precisa e o servidor envia um conjunto fixo de itens daquele recurso, isso pode levar a excesso de itens assim mais dados desnecessários são transferidos, mesmo se outro cliente pedir o mesmo recurso o servidor pode enviar o mesmo recurso pré-montado, pois ele sempre tem os mesmos itens, caso o servidor queira mudar o recurso, eles precisam criar uma versão 2 desse recurso para não quebrar o pedido dos clientes antigos, É o padrão mais fácil de entender e usar, ideal quando a necessidade de dados é fixa. Quando usar REST? para construir serviços web escaláveis e que necessitem de flexibilidade e desempenho. Quando NÃO usar REST? Em casos de comunicação que necessitam de streaming bidirecional em tempo real, como aplicações IoT que exigem comunicação constante e instantânea

**GraphQL:** No GraphQL você usa um formulário de pedido muito detalhado. Você diz exatamente o que quer e o servidor só envia o que você pediu, recebendo só o que é essencial você economiza tempo e banda de internet, principalmente em celulares eliminando o excesso de dados. Como cada pedido é único, o servidor não pode pré-montar um recurso, pois não sabe o que o próximo cliente vai pedir. O cache deve ser feito manualmente, o GraphQL é ideal quando a tela do seu aplicativo precisa de dados muito específicos e variados. Quando usar GraphQL? em aplicações com requisitos de dados complexos e interligados, onde é necessário flexibilidade para buscar exatamente os dados que o cliente precisa, evitando assim requisições excessivas (overfetching) ou insuficientes (underfetching). Quando NÃO usar GraphQL? para projetos pequenos ou simples, quando a API não precisa de flexibilidade complexa, se a complexidade da implementação for um problema, ou para aproveitar ao máximo o cache HTTP

**gRPC:** O gRPC é superior para comunicação inter-serviços em back-end que exige baixa latência e alta performance, além de impor um contrato rigoroso de tipos ele permite que um cliente invoque um método em um servidor remoto como se fosse um objeto local, utilizando HTTP/2 como protocolo de transporte e Protocol Buffers para serialização de dados o gRPC é ideal para comunicação interna de microsserviços de alto desempenho.

Contratos e evolução: Versionamento: Permite que diferentes clientes consumam versões distintas.

Compatibilidade Retroativa: Novas versões funcionam com clientes antigos sem quebrá-los.
Depreciação: Marca um contrato ou funcionalidade como obsoleto
Idempotência: Garante que o efeito de uma operação no sistema seja o mesmo, independentemente de ser executada uma ou várias vezes
Paginação Estável: Mantém a ordem da lista de resultados consistente durante a navegação

Ordenação Determinística: Os resultados da coleção (listas) são sempre ordenados de forma previsível.
Limites de Payload: Restrições de tamanho (máximo) para requisições e respostas

Segurança de ponta a ponta: A segurança de ponta a ponta é garantida por uma arquitetura em camadas onde o OAuth 2.1 rege a autorização, e o OIDC, que utiliza o formato de token JWT, estabelece a identidade, sempre aderindo ao princípio do Privilégio Mínimo (conceder apenas o estritamente necessário, mesmo quando o Escopo é solicitado). Esta arquitetura deve ser robustecida com a Rotação de Chaves para mitigar vazamentos e mecanismos como o Nonce para proteger contra ataques de replay. Na infraestrutura, o TLS Mútuo (mTLS) assegura a autenticação de serviço-para-serviço, enquanto defesas de borda como Rate Limiting e Circuit Breakers atuam contra abuso e falhas em cascata, mitigando riscos críticos como a exposição de PII e o vazamento de secrets através de uma gestão rigorosa.

https://aws.amazon.com/what-is/restful-api/
https://dev.to/cristuker/como-e-porque-usar-graphql-d90

---

Arthur Segalla Rapozo

Qualidade e confiabilidade: proponha uma taxonomia de erros (4xx/5xx, códigos de domínio), correlação de requisições (trace IDs), retry with backoff para operações idempotentes, e políticas de timeouts por integração.

É essencial adotar uma taxonomia clara de erros, separando códigos HTTP (4xx para erros do cliente, 5xx para falhas no servidor) e códigos de domínio específicos para detalhamento adicional. Requisições idempotentes devem implementar retry com backoff exponencial e jitter para evitar sobrecarga em falhas temporárias, cada integração deve ter políticas de timeout bem definidas (conexão, leitura, total), ajustadas com base no SLA do serviço externo, garantindo resiliência sem comprometer a performance.

Observabilidade e SLOs: defina métricas mínimas (latência p50/p95/p99, taxa de erro, saturação), logging estruturado e tracing distribuído. Especifique SLOs plausíveis e suas implicações em capacidade e custo.

É fundamental coletar métricas mínimas como latência (p50, p95, p99), taxa de erro (por tipo e endpoint) e saturação de recursos (CPU, memória, conexões, filas). Logs devem ser estruturados

(JSON, com trace/correlation IDs) para permitir análise automatizada e correlação com métricas e traces.

Desempenho e cache: explique ETags, conditional requests, cache-control, surrogate keys em CDNs, e quando preferir server-driven pagination vs. cursor-based.

Para otimizar desempenho com cache, ETags permitem identificar versões de recursos e habilitar “conditional requests" via “If-None-Match”, reduzindo transferência de dados quando não há mudanças. O cabeçalho Cache-Control define políticas de cache (como “max-age”, “no-cache”, “public/private”) tanto para clientes quanto para intermediários.

---

Gustavo Andre Klahold

Webhooks e eventos: discuta entrega confiável, assinaturas de mensagens, dead-letter queues, reentrega e ordenação. Diferencie event-driven de request/response e explique impacto no design do contrato.

Entrega Confiável: Garante que uma mensagem seja entregue, mesmo em um ambiente instável ou não seguro.
Assinaturas de Mensagens: As mensagens são “Assinadas” para que tenha a confirmação que a mensagem não tenha sido alterada no meio do caminho.
Dead-Letter Queues: As mensagens que tentam ser enviadas muitas vezes mas continuam não sendo entregues elas são enviadas para uma Dead-Letter Queue, que é uma fila especial, especificamente para estas mensagens problemáticas, onde destrava a fila principal, e aí os engenheiros/programadores conseguem ver as mensagens problemáticas e as resolver.
Reentrega: É quando uma entrega não vai, e aí tenta-se fazer uma repetição da entrega dando cada vez mais um tempo maior entre elas.
Ordenação: Ele ordena os pedidos entregues e os pedidos enviados, para que um não chegue antes do outro, normalmente se adiciona um Timestamp e um Sequence ID.

Diferença
Event-Driven: Assíncrona, o controle é feito pelo cliente, normalmente usado para notificações, integrações, orquestração de processos.
Request/Response: Síncrona, o controle é feito pelo publicador, normalmente usado para ações que precisam de resposta imediata

Impacto
No request/response o servidor de da duas operações o GET e o POST, que faz com que o cliente tenha um controle maior, já no event-driven o cliente espera as notificações do publicador, fazendo com que o publicador tenha mais controle.

Governança e DX: descreva o papel de API Gateways, catálogos internos, lint de contrato, revisão técnica, breaking changes policy, changelogs e documentação viva. Inclua critérios de “boa experiência do desenvolvedor”.

API Gateway: Ponto único de entrada para todas as APIs externas.
Catálogos Internos: Catálogo com todas as APIs da organização.
Lint de Contrato: Valida as especificações de API.
Revisão Técnica: Revisão dos design, schema, segurança, performance, códigos e mensagens padronizados, e um versionamento.
Breaking Changes Policy: Gestão de mudanças.
Changelogs: Registro cronológico das mudanças significativas feitas.
Documentação Viva: Documentação que é gerada automaticamente a partir do código fonte e atualizada em tempo real.

Critérios de "Boa Experiência do Desenvolvedor"
Uma Documentação de Qualidade, Consistência e Previsibilidade nas APIs e no código, Performance e Confiabilidade no código.

---

Heitor Pereira Emmerich

Integração com PHP: mapeie como clientes PHP consomem APIs com foco em resiliência (ex.: retries idempotentes, timeouts, connection pools), proteção de segredos e padronização de clients internos.

Como clientes PHP na hora de consumir APIs você tem diversas modos de realizar esta ação com resiliência ou seja fazer do melhor jeito possível e do jeito que faça a API não cair ou ter problemas, dentro desses existem retries idempotentes, timeouts e connection pools, que respectivamente funcionam assim:
o primeiro é basicamente uma funcionalidade que utiliza-se PUT e DELETE nas requisições do php que realizam o trabalho para sempre que haja um request do mesmo recurso sem necessidade ele verificará se realmente já tem algo com o mesmo request e impedirá sem duplicar o request fazendo assim o request funcionar porém sem criar problemas de duplicação
o segundo é um tempo limite que o servidor terá de espera para a resposta da api, se ocorrer dela não ser respondida a tempo o sistema pára de esperar e para de tentar fazer esse pedido para evitar travamentos, o terceiro servem para reusar conexões que já estão sendo usadas pelo servidor deixando mais leve e descartando a necessidade de realizar múltiplas conexões novas para as mesmas requisições. a proteção de dados e mais deixar informações sensíveis fora do codigo público, e por fim padronizar os clients internos ou seja tipo botar um molde nas requisições e respostas para facilitar a manutenção e integração em outros sistemas

Compliance e custos: aponte riscos legais (LGPD/PII, retenção de logs), políticas de data minimization, auditoria e estimativa de custo ao integrar APIs de terceiros (preço por chamada, limites, fallback e graceful degradation).

normalmente APIs de terceiros tem seus riscos principalmente por normalmente terem contato com informações sensíveis do usuários caindo no cuidado para não quebrar o LGPD e PII e garantir que tenha consentimento,  segurança e políticas claras de guardar e exclusão de logs. deve também aplicar o princípio de data minimization, coletando apenas as informações estritamente que são necessárias e armazenando elas pelo menor tempo possível. além disso auditorias periódicas devem verificar a conformidade de normas de privacidade e segurança. ja no aspecto financeiro é importante estimar custos por chamada, limites de uso e estratégias de fallback e graceful degradation caso haja problema de queda sobre o provedor

---

Referências


https://dev.to/asouza/entendendo-a-idempotencia-e-resiliencia-em-apis-23e2
https://medium.com/@marcelomg21/event-driven-architecture-eda-em-uma-arquitetura-de-micro-servi%C3%A7os-1981614cdd45
https://www.duplod.com.br/o-que-e-o-request-e-response/