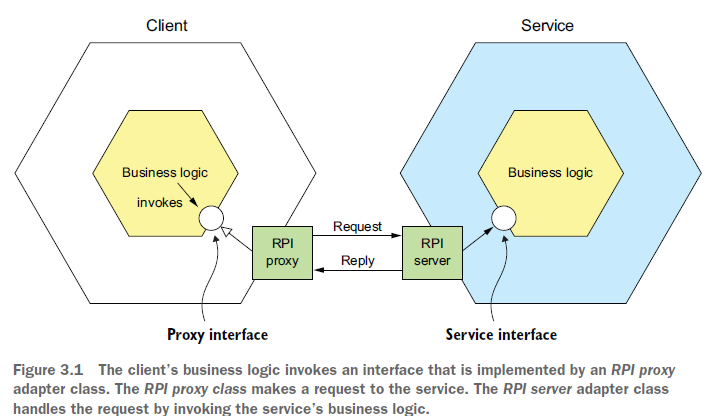
Synchronous Remote Procedure

Invocation (RPI) in Microservices

In RPI, A **client invokes a service** using a **synchronous,** **remote procedure invocation-based protocol, such as REST.**

When using a remote procedure invocation-based IPC mechanism, a client sends a request to a service, and the service processes the request and sends back a response**. Some clients may block waiting** for a response, and **others might have a reactive, nonblocking architecture**. But **unlike when using messaging, the client assumes that the response will arrive in a timely fashion.**



**The *proxy interface* usually encapsulates the underlying communication protocol**.

There are numerous protocols to choose from. We’ll talk about REST and gRPC as examples and how to improve the availability of your services by properly handling partial failure and **explain why a microservices-based application that uses RPI must use a service discovery mechanism.**

# Using REST

*REST* is an IPC mechanism that **(almost always) uses HTTP.** Roy Fielding, the creator of REST, defines REST as follows:

**(You should later on look for how exactly each one of these benefits are fulfilled:**

<https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design>

<https://www.ibm.com/topics/rest-apis>

<https://en.wikipedia.org/wiki/REST>

).

“*REST provides a set of architectural constraints that, when applied as a whole, emphasizes* ***scalability of component interactions****,* ***generality of interfaces, independent deployment of components****,* ***and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems*.”**

* A key concept in REST is a ***resource***, **which typically represents a single business object, such as a Customer or Product, or a collection of business objects.**
* **REST uses the HTTP verbs for manipulating resources**, **which are referenced using a URL:**

For example, a GET request returns the **representation of a resource**, which is **often** in the form of **an XML document or JSON object**, although **other formats such as binary can be used.** A POST request creates a new resource, and a PUT request updates a resource. The Order Service, for example, has a POST /orders endpoint for creating an Order and a GET /orders/{orderId} endpoint for retrieving an Order.

Many developers claim their HTTP-based APIs are RESTful. But as Roy Fielding describes in a blog post**, not all of them actually are** (<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>). To understand why, let’s take a look at the **REST maturity model:**

## The REST Maturity Model

Leonard Richardson (no relation to your author) defines a very useful maturity model for REST (http://martinfowler.com/articles/richardsonMaturityModel.html) that consists of the following levels:

 *Level 0*—Clients of a level 0 service invoke the service by making HTTP POST requests to its **sole URL endpoint.** Each request specifies the action to perform, the target of the action (for example, the business object), and any parameters.(single verb/single URL)

I add: on this level you’re basically using the HTTP as a transport tool to just invoke a remote procedure and you’re not using it as a platform to its fullest potential (verbs, status codes, etc.). and at this level they use **a single verb** that’s usually POST and the request details come in the body of the request or GET with parameters and actions in the URL, and even the errors like 400 will be sent by 200ok announcing the error in the response body.

 *Level 1*—**A level 1 service supports the idea of resources**. Level one employs **many URIs but only a single HTTP verb To perform an action on a resource**.(single verb/ multiple URLs)

a client for example makes a POST request that specifies the action to perform and any parameters in the body, or Get request with actions and parameters in the URL.

 *Level 2*—**A level 2 service uses HTTP verbs to perform actions**: GET to retrieve, POST to create, and PUT to update. **The request query parameters and body**, if any, specify the **actions' parameters**.(multiple URLs/multiple verbs/multiple status codes/even location header in a 201ok response pointing to where the new resource can be retrieved)

**This enables services to use web infrastructure such as caching for GET requests.**

 *Level 3*—The design of a level 3 service is based on the terribly named HATEOAS (Hypertext As The Engine Of Application State) principle. **The basic idea is that the representation of a resource returned by a GET request contains links for performing actions on that resource**. For example, a client can cancel an order using a link in the representation returned by the GET request that retrieved the order(They can be sent as a response to a POST request too. Check out the below example). The benefits of HATEOAS **include no longer having to hard-wire URLs into client code** (<www.infoq.com/news/2009/04/hateoas-restful-api-advantages>). As an example of a response to a GET request for a list of open slots:

<openSlotList>

<slot id = "1234" doctor = "mjones" start = "1400" end = "1450">

<link rel = "/linkrels/slot/book"

uri = "/slots/1234"/>

</slot>

<slot id = "5678" doctor = "mjones" start = "1600" end = "1650">

<link rel = "/linkrels/slot/book"

uri = "/slots/5678"/>

</slot>

</openSlotList>

Each slot now has a link element which contains a URI to tell us how to book an appointment. The point of hypermedia controls is that **they tell us what we can do next**, **and the URI of the resource we need to manipulate to do it**. Rather than us having to know where to post our appointment request, the hypermedia controls in the response tell us how to do it:



* One obvious benefit of hypermedia controls is that it allows the server to change its URI scheme without breaking clients. As long as clients look up the “addTest” link URI then the server team can juggle all URIs other than the initial entry points.

### The meaning of the Levels

I should stress that the RMM, while a good way to think about what the elements of REST are, is not a definition of levels of REST itself.

Roy Fielding has made it clear that Level 3 RMM(Richardson Maturity Model) is a pre-condition for REST APIs meaning that REST APIs must be hypertext-driven. You can see the levels as **tool to help us learn about the concepts** and **not something that should be used in some kind of** **assessment mechanism:**

* **Level 1** tackles the question of **handling complexity** by using **divide and conquer,** breaking a large service endpoint down into multiple resources.
* **Level 2** introduces a standard set of verbs so that **we handle similar situations in the same way, removing unnecessary variation.**
* **Level 3 introduces discoverability**, providing a way of making a protocol **more self-documenting**.

## Specifying REST APIs

As mentioned earlier in section 3.1, **you must define your APIs using an interface definition language** (IDL). Unlike older communication protocols like CORBA and SOAP, REST did not originally have an IDL. Fortunately, the developer community has rediscovered the value of an IDL for RESTful APIs.

The most popular **REST IDL is the Open API Specification** (www.openapis.org), which evolved from the Swagger open-source project. The Swagger project is a set of tools for developing and documenting REST APIs. It includes tools that generate client stubs and server skeletons **from an interface definition.**

## The Challenge of Fetching Multiple Resources in a Single Request

REST resources are usually oriented around business objects, such as Consumer and Order. Consequently, a common problem when designing a REST API is how to enable the client to retrieve multiple related objects in a single request. For example, imagine that a REST client wanted to retrieve an Order and the Order's Consumer. **A pure REST API would require the client to make at least two requests, one for the Order and another for its Consumer**. A more complex scenario would require even more round-trips and suffer from excessive latency.

One solution to this problem is for an API to allow the client to retrieve related resources when it gets a resource. For example, a client could retrieve an Order and its Consumer using GET /orders/order-id-1345**?expand=consumer**:

* The query parameter specifies the related resources to return with the Order.
* This approach **works well in many scenarios** but it’s often **insufficient for more complex scenarios**
* It’s also potentially time consuming to implement.

**This has led to** the increasing popularity of alternative API technologies such as **GraphQL** (http://graphql.org) and **Netflix Falcor** (http://netflix.github.io/falcor/), which are designed to support efficient data fetching.

## The Challenge of Mapping Operations to HTTP Verbs

Another common REST API design problem is how to map the operations you want to perform on a business object to an HTTP verb.

A REST API should use PUT for updates, **but there may be multiple ways to update an order**, including **cancelling it, revising the order**, and so on.

Also, **an update might not be idempotent**, **which is a requirement for using PUT**.

* One solution is to define a **sub-resource for updating a particular aspect of a resource**. The Order Service, for example, has a POST “/orders/ {orderId}/cancel” endpoint for cancelling orders, and a POST “/orders/{orderId}/ revise” endpoint for revising orders.
* Another solution is to **specify a verb as a URL query parameter.**
* Sadly, **neither solution is particularly RESTful.**

**This problem with mapping operations to HTTP verbs has led to the growing popularity of alternatives to REST, such as gPRC**, discussed shortly in section 3.2.2.

But first let’s look at the benefits and drawbacks of REST.

## Benefits and Drawbacks of REST

There are numerous benefits to using REST:

 It’s simple and familiar.

 You can test an HTTP API from within a browser using, for example, the Postman plugin, or from the command line using curl (assuming JSON or some other text format is used).

 It directly supports request/response style communication.

 **HTTP is, of course, firewall friendly.**

 **It doesn’t require an intermediate broker**, which simplifies the system’s architecture.

There are some drawbacks to using REST:

** It only supports the request/response style of communication.**

 **Reduced availability**. Because the client and service communicate directly **without an intermediary to buffer messages,** they must both be running for the duration of the exchange.

 Clients must know the locations (URLs) of the service instances(s). As described in section 3.2.4, this is a nontrivial problem in a modern application. Clients must use what is known as a *service discovery mechanism* to locate service instances.

 **Fetching multiple resources in a single request is challenging.**

 **It’s sometimes difficult to map multiple update operations to HTTP verbs.**

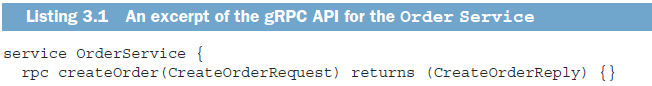
Despite these drawbacks, **REST seems to be the de facto standard for APIs**, though there are a couple of interesting alternatives. GraphQL, for example, implements **flexible, efficient data fetching**.

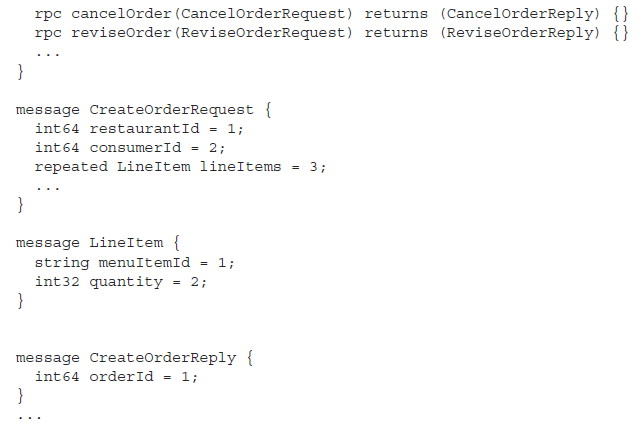
Chapter 8 discusses GraphQL and covers the API gateway pattern.

gRPC is another alternative to REST. Let’s take a look at how it works.

# Using gRPC

* As mentioned in the preceding section, one challenge with using REST is that because HTTP only provides **a limited number of verbs**, it’s **not** always **straightforward** **to design a REST** API **that supports multiple update operations**. An IPC technology that avoids this issue is gRPC (<www.grpc.io>), a framework for writing cross-language clients and servers (see [https://en.wikipedia.org/wiki/Remote\_procedure\_call](https://en.wikipedia.org/wiki/Remote_procedure_call%20) for more).
* **gRPC is a binary message-based protocol**, and this means—as mentioned earlier in the discussion of binary message formats—**you’re forced to take an API-first approach to service design.**
* You define your gRPC APIs using a Protocol Buffers-based IDL, which is Google’s language-neutral mechanism for serializing structured data. You use the **Protocol Buffer compiler** to generate **client-side stubs** and **server-side skeletons**. The compiler can generate code for a variety of languages, including Java, C#, NodeJS, and GoLang. Clients and servers exchange binary messages in the Protocol Buffers format **using HTTP/2.**
* A gRPC API consists of **one or more services** **and request/response message definitions**. **A *service definition* is analogous to a Java interface and is a collection of strongly typed methods.**
* As well as supporting simple request/response RPC, **gRPC support streaming RPC**. **A server can reply with a stream of messages to the client**. Alternatively, **a client can send a stream of messages** to the server.
* gRPC uses Protocol Buffers as the message format. Protocol Buffers is, as mentioned earlier, an efficient, compact, binary format. It’s a tagged format. Each field of a Protocol Buffers message is numbered and has a type code. A message recipient can extract the fields that it needs and skip over the fields that it doesn’t recognize. As a result, gRPC enables APIs to evolve while remaining backward-compatible.
* Listing 3.1 shows an excerpt of the gRPC API for the Order Service. It defines several methods, including createOrder(). This method takes a CreateOrderRequest as a parameter and returns a CreateOrderReply: (=2 for example defines the **field tag**)





* gRPC has several benefits:

 It’s straightforward to design an API that has **a rich set of update operations.**

 It has an efficient, compact IPC mechanism, especially when exchanging large messages.

 Bidirectional streaming enables **both RPI and messaging styles of communication**.

 It enables interoperability between clients and services written in a wide range of languages.

* gRPC also has several drawbacks:

 It takes more work for JavaScript clients to consume gRPC-based API than REST/JSON-based APIs.

 Older firewalls might not support HTTP/2.

gRPC is a compelling alternative to REST, but like REST, it’s **a synchronous communication mechanism, so it also suffers from the problem of partial failure.** Let’s take a look at what that is and how to handle it.

# Handling Partial Failure Using the Circuit Breaker Pattern

Did u figure out what exactly synchronous mean/reactive/asynchronous?