You can also download the complete set of articles, plus information about implementing microservices using NGINX Plus, as an ebook – Microservices: From Design to Deployment. Also, please look at the new Microservices Solutions page.

This is the third article in our series about building applications with a microservices architecture. The first article introduces the Microservices Architecture pattern, compares it with the Monolithic Architecture pattern, and discusses the benefits and drawbacks of using microservices. The second article describes how clients of an application communicate with the microservices via an intermediary known as an API Gateway. In this article, we take a look at how the services within a system communicate with one another. The fourth article explores the closely related problem of service discovery.

Introduction

In a monolithic application, components invoke one another via language-level method or function calls. In contrast, a microservices-based application is a distributed system running on multiple machines. Each service instance is typically a process. Consequently, as the following diagram shows, services must interact using an inter-process communication (IPC) mechanism.

- One-to-many – Each request is processed by multiple service instances.

The second dimension is whether the interaction is synchronous or asynchronous:

- Synchronous – The client expects a timely response from the service and might even block while it waits.
- Asynchronous – The client doesn't block while waiting for a response, and the response, if any, isn't necessarily sent immediately.

The following table shows the various interaction styles.

	One-to-One	**One-to-Many**
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

There are the following kinds of one-to-one interactions:

- Request/response – A client makes a request to a service and waits for a response. The client expects the response to arrive in a timely fashion. In a thread-based application, the thread that makes the request might even block while waiting.
- Notification (a.k.a. a one-way request) – A client sends a request to a service but no reply is expected or sent.
- Request/async response – A client sends a request to a service, which replies asynchronously. The client does not block while waiting and is designed with the assumption that the response might not arrive for a while.

There are the following kinds of one-to-many interactions:

- Publish/subscribe – A client publishes a notification message, which is consumed by zero or more interested services.

The services use a combination of notifications, request/response, and publish/subscribe. For example, the passenger's smartphone sends a notification to the Trip Management service to request a pickup. The Trip Management service verifies that the passenger's account is active by using request/response to invoke the Passenger Service. The Trip Management service then creates the trip and uses publish/subscribe to notify other services including the Dispatcher, which locates an available driver.

Now that we have looked at interaction styles, let's take a look at how to define APIs.

Defining APIs

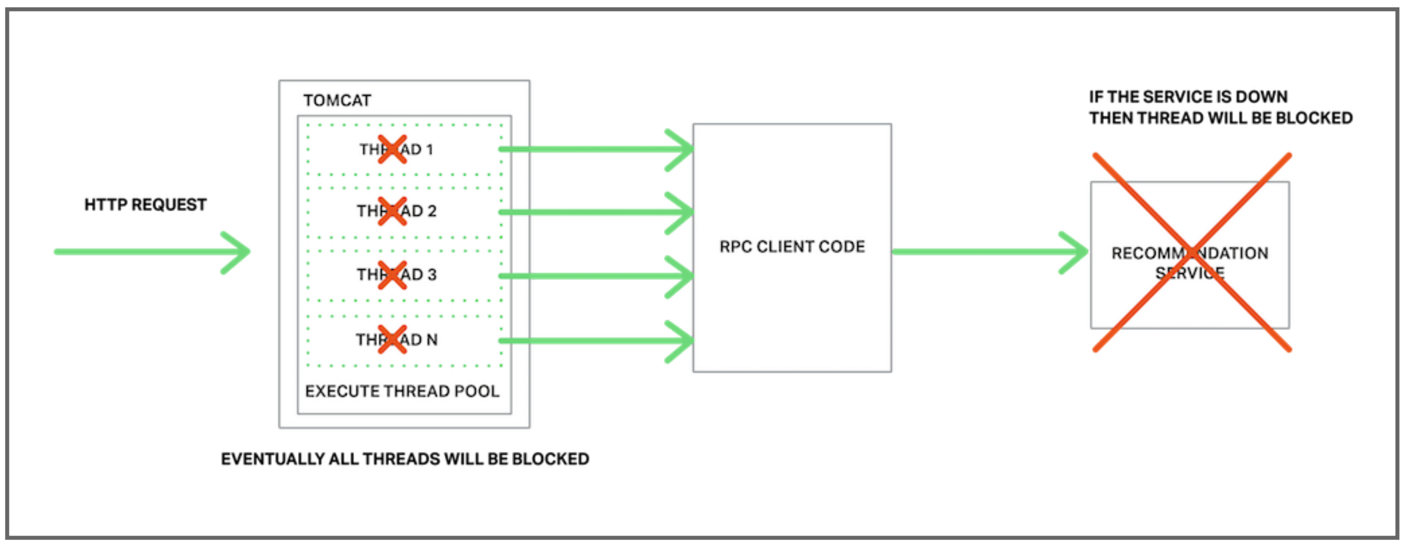
A service's API is a contract between the service and its clients. Regardless of your choice of IPC mechanism, it's important to precisely define a service's API using some kind of interface definition language (IDL). There are even good arguments for using an API-first approach to defining services. You begin the development of a service by writing the interface definition and reviewing it with the client developers. It is only after iterating on the API definition that you implement the service. Doing this design up front increases your chances of building a service that meets the needs of its clients.

As you will see later in this article, the nature of the API definition depends on which IPC mechanism you are using. If you are using messaging, the API consists of the message channels and the message types. If you are using HTTP, the API consists of the URLs and the request and response formats. Later on we will describe some IDLs in more detail.

Evolving APIs

A service's API invariably changes over time. In a monolithic application it is usually straightforward to change the API and update all the callers. In a microservices-based application it is a lot more difficult, even if all of the consumers of your API are other services in the same application. You requests.

Consider, for example, the Product details scenario from that article. Let's imagine that the Recommendation Service is unresponsive. A naive implementation of a client might block indefinitely waiting for a response. Not only would that result in a poor user experience, but in many applications it would consume a precious resource such as a thread. Eventually the runtime would run out of threads and become unresponsive as shown in the following figure.



To prevent this problem, it is essential that you design your services to handle partial failures.

A good to approach to follow is the one described by Netflix. The strategies for dealing with partial failures include:

- Network timeouts – Never block indefinitely and always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.
- Limiting the number of outstanding requests – Impose an upper bound on the number of outstanding requests that a client can have with a particular service. If the limit has been reached, it is probably pointless to make additional requests, and those attempts need to fail immediately.

When using messaging, processes communicate by asynchronously exchanging messages. A client makes a request to a service by sending it a message. If the service is expected to reply, it does so by sending a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. Instead, the client is written assuming that the reply will not be received immediately.

A message consists of headers (metadata such as the sender) and a message body. Messages are exchanged over channels. Any number of producers can send messages to a channel. Similarly, any number of consumers can receive messages from a channel. There are two kinds of channels, point-to-point and publish-subscribe. A point-to-point channel delivers a message to exactly one of the consumers that is reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier. A publish-subscribe channel delivers each message to all of the attached consumers. Services use publish-subscribe channels for the one-to-many interaction styles described above.

The following diagram shows how the taxi-hailing application might use publish-subscribe channels.

STOMP. Other messaging systems have proprietary but documented protocols. There are a large number of open source messaging systems to choose from, including RabbitMQ, Apache Kafka, Apache ActiveMQ, and NSQ. At a high level, they all support some form of messages and channels. They all strive to be reliable, high-performance, and scalable. However, there are significant differences in the details of each broker's messaging model.

There are many advantages to using messaging:

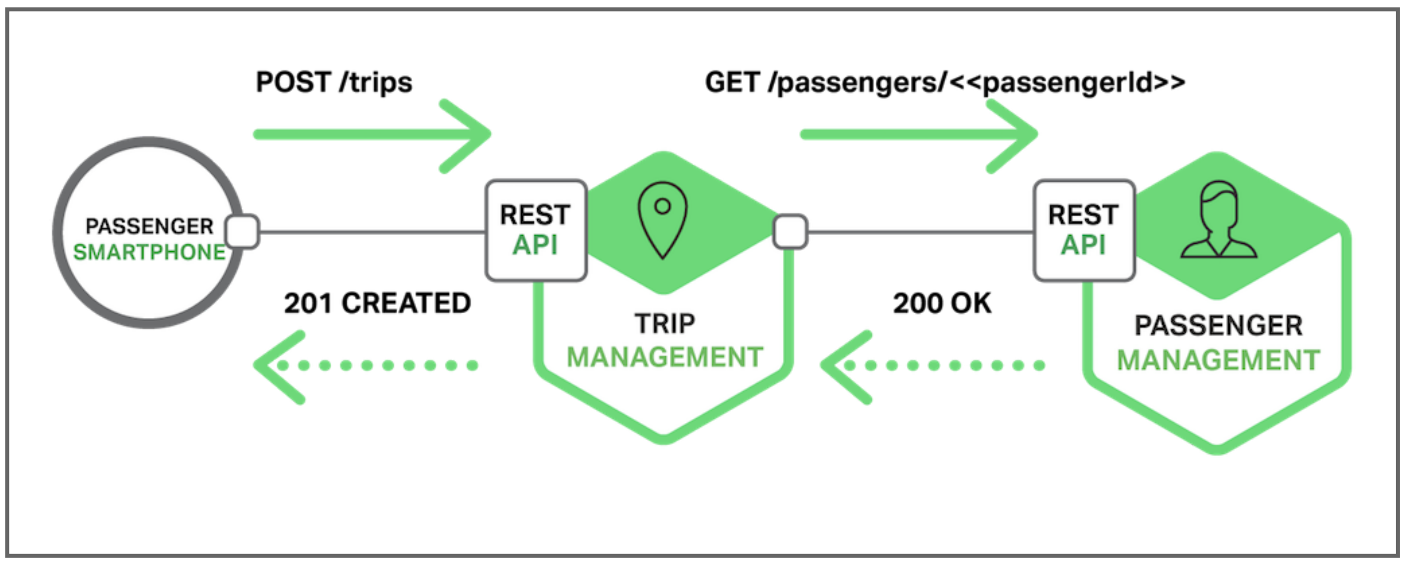
- Decouples the client from the service – A client makes a request simply by sending a message to the appropriate channel. The client is completely unaware of the service instances. It does not need to use a discovery mechanism to determine the location of a service instance.
- Message buffering – With a synchronous request/response protocol, such as a HTTP, both the client and service must be available for the duration of the exchange. In contrast, a message broker queues up the messages written to a channel until they can be processed by the consumer. This means, for example, that an online store can accept orders from customers even when the order fulfillment system is slow or unavailable. The order messages simply queue up.
- Flexible client-service interactions – Messaging supports all of the interaction styles described earlier.
- Explicit inter-process communication – RPC-based mechanisms attempt to make invoking a remote service look the same as calling a local service. However, because of the laws of physics and the possibility of partial failure, they are in fact quite different. Messaging makes these differences very explicit so developers are not lulled into a false sense of security.

There are, however, some downsides to using messaging:

- Additional operational complexity – The messaging system is yet another system component that must be installed, configured, and operated. It's essential that the message broker be highly available, otherwise system reliability is impacted.
- Complexity of implementing request/response-based interaction – Request/response-style interaction requires some work to implement. Each request message must contain a reply channel

> *"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."*
>
> – Fielding, Architectural Styles and the Design of Network-based Software Architectures

The following diagram shows one of the ways that the taxi-hailing application might use REST.



request sent to retrieve the order. Benefits of HATEOAS include no longer having to hardwire URLs into client code. Another benefit is that because the representation of a resource contains links for the allowable actions, the client doesn't have to guess what actions can be performed on a resource in its current state.

There are numerous benefits to using a protocol that is based on HTTP:

- HTTP is simple and familiar.
- You can test an HTTP API from within a browser using an extension such as Postman 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 HTTP:

- It only directly supports the request/response style of interaction. You can use HTTP for notifications but the server must always send an HTTP response.
- Because the client and service communicate directly (without an intermediary to buffer messages), they must both be running for the duration of the exchange.
- The client must know the location (i.e., the URL) of each service instance. As described in the previous article about the API Gateway, this is a non-trivial problem in a modern application. Clients must use a service discovery mechanism to locate service instances.

The developer community has recently rediscovered the value of an interface definition language for RESTful APIs. There are a few options, including RAML and Swagger. Some IDLs such as Swagger allow you to define the format of request and response messages. Others such as RAML require you to use a separate specification such as JSON Schema. As well as describing APIs, IDLs typically have tools that generate client stubs and server skeletons from an interface definition.

microservices in a single language today, it's likely that you will use other languages in the future.

There are two main kinds of message formats: text and binary. Examples of text-based formats include JSON and XML. An advantage of these formats is that not only are they human-readable, they are self-describing. In JSON, the attributes of an object are represented by a collection of name-value pairs. Similarly, in XML the attributes are represented by named elements and values. This enables a consumer of a message to pick out the values that it is interested in and ignore the rest. Consequently, minor changes to the message format can be easily backward compatible.

The structure of XML documents is specified by an XML schema. Over time the developer community has come to realize that JSON also needs a similar mechanism. One option is to use JSON Schema, either stand-alone or as part of an IDL such as Swagger.

A downside of using a text-based message format is that the messages tend to be verbose, especially XML. Because the messages are self-describing, every message contains the name of the attributes in addition to their values. Another drawback is the overhead of parsing text. Consequently, you might want to consider using a binary format.

There are several binary formats to choose from. If you are using Thrift RPC, you can use binary Thrift. If you get to pick the message format, popular options include Protocol Buffers and Apache Avro. Both of these formats provide a typed IDL for defining the structure of your messages. One difference, however, is that Protocol Buffers uses tagged fields, whereas an Avro consumer needs to know the schema in order to interpret messages. As a result, API evolution is easier with Protocol Buffers than with Avro. This blog post is an excellent comparison of Thrift, Protocol Buffers, and Avro.

Summary