Making retries safe with idempotent APIs

Malcolm Featonby

Making retries safe with idempotent APIs

Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved

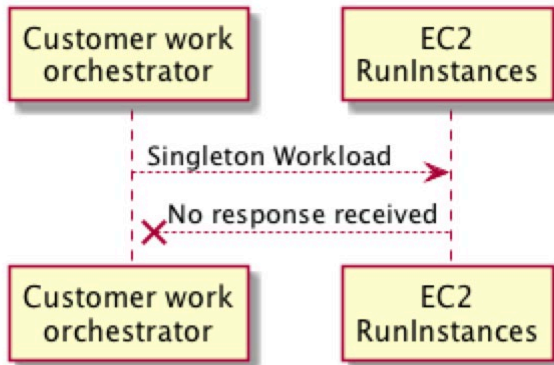
At Amazon, we often see patterns in our services in which a complex operation is decomposed into a controlling process making calls to a number of smaller services, each responsible for one part of the overall workflow. For example, consider the launch of an Amazon Elastic Compute Cloud (EC2) instance. “Under the hood” this involves calls to services responsible for making placement decisions, creating Amazon Elastic Block Store (EBS) volumes, creating elastic network Interfaces, and provisioning a virtual machine (VM). On occasion, one or more of these service calls might fail. To deliver a good customer experience we want to drive this workflow to success. To achieve this, we look to the controlling process to shepherd all decomposed services into a known good state.

We’ve found that in many cases the simplest solution is the best solution. In the scenario I just described, it would be best to simply retry these calls until they succeed. Interestingly, as Marc Brooker explains in the article Timeouts, retries, and backoff with jitter, a surprisingly large number of transient or random faults can be overcome by simply retrying the call. As simple as it seems, this pattern has proven so effective that we have baked default retry behavior into some of our AWS SDK implementations. These implementations automatically retry requests that fail as a result of network IO issues, server-side fault, or service rate limiting. Being able to simply retry requests reduces the number of edge cases that need to be dealt with by the client. As a result, it reduces the amount of undifferentiated boilerplate code needed in calling services. Undifferentiated boilerplate code in this context refers to the code needed to wrap service calls to remote services to handle various fault scenarios that might arise.

However, retrying a service call as mitigation for a transient fault is based on a simplifying assumption that an operation can be retried without any side effects. Put another way, we want to make sure that the result of the call happens only once, even if we need to make that call multiple times as part of our retry loop. Going back to our earlier example, it would be undesirable for the EC2 instance launch workflow to retry a failed call to create an EBS volume and end up with two EBS volumes. In this article, we discuss how AWS leverages idempotent API operations to mitigate some of the potential undesirable side effects of retrying in order to deliver a more robust service while still leveraging the benefits of retries to simplifying client-side code.

Retrying and the potential for undesirable side effects

To dive into this more deeply let’s consider a hypothetical scenario where a customer is using the Amazon EC2 RunInstances API operation. In our scenario, our customer wants to run a *singleton* workload, which is a workload that requires “at most one” EC2 instance running at any time. To achieve this, our customer’s provisioning process asks Amazon EC2 to launch this new workload. However, for some reason, perhaps due to a network timeout, the provisioning process receives no response.



This leaves the provisioning process with a dilemma. It's not clear whether the singleton workload is running or not. Simply retrying the request could result in multiple workloads, which could have dire consequences. To overcome this dilemma the provisioning process has to perform a reconciliation to determine whether this workload is running or not. This is a lot of heavy lifting to compensate for an edge case that might happen relatively infrequently. In addition, even in the case of a reconciliation workflow, there might still be some uncertainty. What if the resource is there but created by another provisioning process? In the simple case that might be fine, but in more complex scenarios it might be important to know whether the resource was created by this process or another process.

Reducing client complexity with idempotent API design

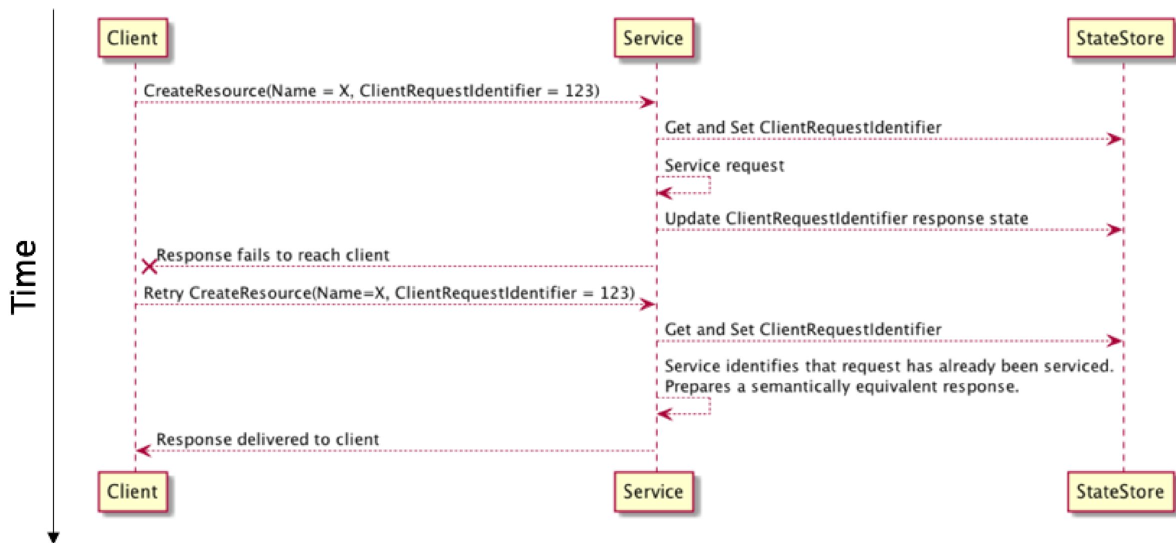
To allow callers to retry these kinds of operations we need to make them idempotent. An *idempotent* operation is one where a request can be retransmitted or retried with no additional side effects, a property that is very beneficial in distributed systems.

We can significantly simplify client code by delivering a contract that allows the client to make a simplifying assumption that any error that isn't a validation error can be overcome by retrying the request until it succeeds. However, this introduces some additional complexity to service implementation. In a distributed system with many clients making many calls and with many requests in flight, the challenge is how do we identify that a request is a repeat of some previous request?

Many approaches could be used to infer whether a request is a duplicate of an earlier request. For example, it might be possible to derive a synthetic token based on the parameters in the request. You could derive a hash of the parameters present and assume that any request from the same caller with identical parameters is a duplicate. On the surface, this seems to simplify both the customer experience and the service implementation. Any request that looks exactly like a previous request is considered a duplicate. However, we have found that this approach doesn't work in all cases. For example, it might be reasonable to assume that two exact duplicate requests from the same caller to create an Amazon DynamoDB table received very close together in time are duplicates of the same request. However, if those requests were for launching an Amazon EC2 instance, then our assumption might not hold. It's possible that the caller actually wants two identical EC2 instances.

At Amazon, our preferred approach is to incorporate a unique caller-provided client request identifier into our API contract. Requests from the same caller with the same client request identifier can be considered duplicate requests and can be dealt with accordingly. By allowing customers to clearly express intent through API semantics we want to reduce the potential for unexpected outcomes for the customer. A unique caller-provided client request identifier for idempotent operations meets this need. It also has the benefit of making that intent readily auditable because the unique identifier is present in logs like AWS CloudTrail. Furthermore, by labeling the created resource with the unique client request identifier, customers are able to identify resources created by any given request. A concrete example of this can be seen in the Amazon EC2 DescribeInstances response, which shows the unique identifier used to create the EC2 instance. (In the Amazon EC2 API the unique client request identifier is called the ClientToken).

The following diagram shows a sample request/response flow that uses a unique client request identifier in an idempotent retry scenario:



In this example, a customer requests the creation of a resource that presents a unique client request identifier. On receiving the request, the service first checks to see if it has seen this identifier before. If it has not, it starts to process the request. It creates an idempotent "session" for this request keyed off the customer identifier and their unique client request identifier. If a subsequent request is received from the same customer with the same unique client request identifier then the service knows it has already seen this request and can take appropriate action. An important consideration is that the process that combines recording the idempotent token and all mutating operations related to servicing the request must meet the properties for an atomic, consistent, isolated, and durable (ACID) operation. An ACID server-side operation needs to be an "all or nothing" or atomic process. This ensures that we avoid situations where we could potentially record the idempotent token and fail to create some resources or, alternatively, create the resources and fail to record the idempotent token.

The previous diagram shows the preparation of a semantically equivalent response in cases where the request has already been seen. It could be argued that this is not required to meet the letter of the law for an operation to be idempotent. Consider the case where a hypothetical CreateResource operation is called with the unique request identifier123. If the first request is received and processed but the response never makes it back to the caller then the caller will retry the request with identifier 123. However, the resource might now have been created as part of the initial request. One possible response to this request is to return a ResourceAlreadyExists return code. This meets the basic tenets for idempotency because there is no side effect for retrying the call. However, this leads to uncertainty from the perspective of the caller because it's not clear whether the resource was created as a result of this request or the resource was created as the result of an earlier request. It also makes introducing retry as the default behavior a little more challenging. This is because, although the request had no side effects, the subsequent retry and resultant return code will likely change the flow of execution for the caller. Now the caller needs to deal with the resource already existing even in cases where (from their perspective) it did not exist before they made the call. In this scenario, although there is no side effect from the service perspective, returning ResourceAlreadyExists has a side effect from the client's perspective.

Semantic equivalence and support for default retry strategies

An alternative is to deliver a semantically equivalent response in every case for the same unique request identifier for some interval. This means that any subsequent response to a retry request from the same caller with the same unique client request identifier will have the same meaning as the first response returned for the first successful request. This approach has some really useful properties—especially where we want to improve the customer experience by safely and simply retrying operations that experience server-side faults, just as we do with the AWS SDK through default retry policies.

We can see an example of idempotency with semantically equivalent responses and automated retry logic in action when we use the Amazon EC2 RunInstances API operation and the AWS Command Line Interface (CLI). Note that the AWS CLI (like the AWS SDK) supports a default retry policy, which we are using here. In this example, we launch an EC2 instance using the following AWS CLI command:

```
$ aws ec2 run-instances --image-id ami-04fcd96153cb57194 --instance-type t2.micro
{
   "Instances": [
      {
         "Monitoring": {
            "State": "disabled"
         },
         "StateReason": {
            "Message": "pending",
            "Code": "pending"
         },
         "State": {
```

```
            "Code": 0,
            "Name": "pending"
        },
        "InstanceId": "i-xxxxxxxxxxxxxxxxx",
        "ImageId": "ami-04fcd96153cb57194",
        …
        "ClientToken": "eb3c3141-a229-4ca0-b005-eb922e2cabdc",
        …
```

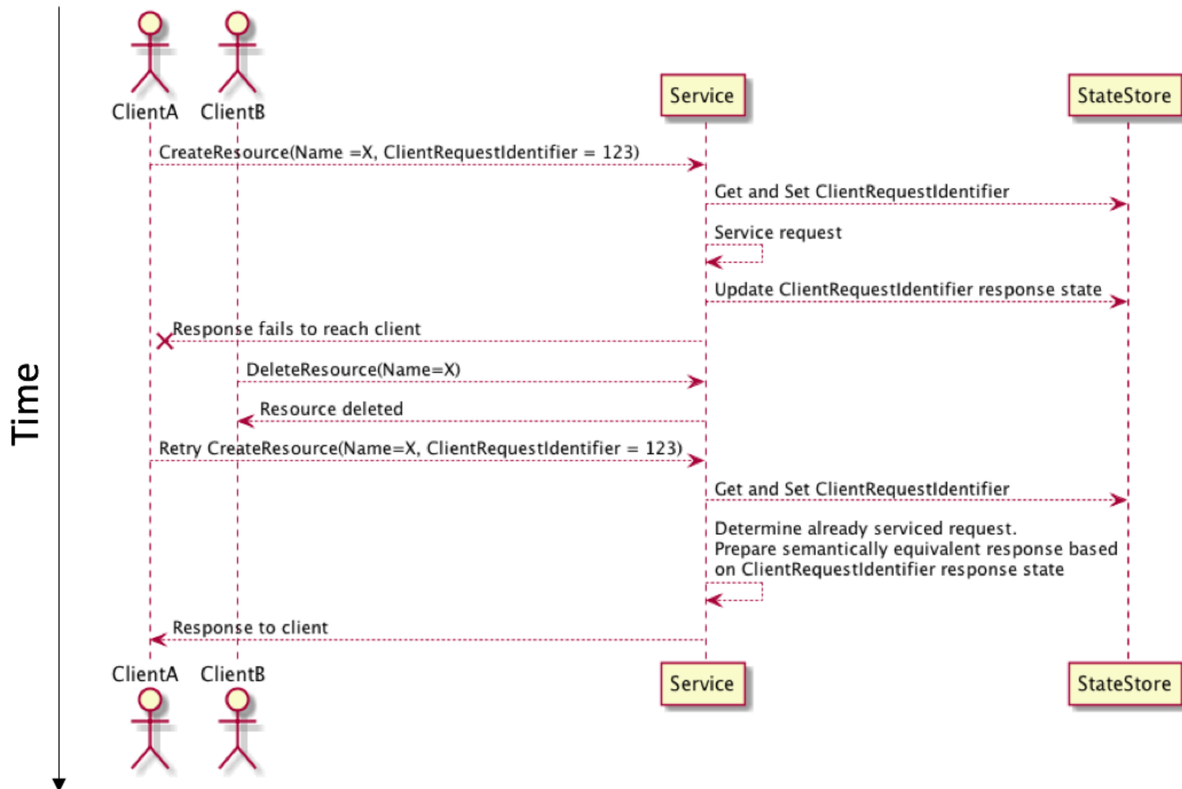
In many AWS API operations, the unique client request identifier is modeled by the ClientToken field. Note that we didn't provide a unique client request identifier in our request. The ClientToken property in the response is echoed back by the remote service. This is because the AWS CLI will generate a unique ID for the request if an identifier is not provided, which allows retries to happen "under the hood" in the event of a transient upstream service fault. An example of what a retried request/response might look like can be simulated by waiting a few minutes and then making another request using the AWS CLI and the same ClientToken that was returned in the previous request. It's important to note that the response returned is very similar but not identical to the first response:

```
$ aws ec2 run-instances --image-id ami-04fcd96153cb57194 --instance-type t2.micro --client-token
eb3c3141-a229-4ca0-b005-eb922e2cabdc
{
    "Instances": [
        {
            "Monitoring": {
                "State": "disabled"
            },
            "StateReason": {
                "Message": "running", ← Now running
                "Code": "running"  ← Now running
            },
            "State": {
                "Code": 16,  ← Now running status code
                "Name": "running"  ← Now running
            },
            "InstanceId": "i-xxxxxxxxxxxxxxxxx",
            "ImageId": "ami-04fcd96153cb57194",
            …
            "ClientToken": "eb3c3141-a229-4ca0-b005-eb922e2cabdc",
            …
```

The AWS SDK and AWS CLI leverage this behavior whenever it's available to simplify the experience of the customer using these tools. Our SDK and CLI are aware of which operations support an idempotent contract, and they will generate a unique client request identifier and add it to the request if one is not provided by the caller. This generated identifier is then reused in the event of a retry, thus ensuring we meet the "at most once" commitment for the request. Because the response (even on a retry) is semantically equivalent, the calling client code can be completely unaware of any retries that happen within the SDK code and simply deal with the response when it is received.

Late arriving requests and the life span of unique client request identifiers

An interesting edge case with idempotent solutions in distributed systems can occur when requests arrive late. Consider the hypothetical situation where we have two actors in a service and a scenario where the client creating the resource retries the operation and the subsequent request is delayed. In the interim a second actor deletes the resource:



The best way to deal with this scenario can vary from service to service. The approach we have taken for EC2 RunInstances is to honor the initial idempotent contract even in the scenario we just described. In this situation we hold with the principle of least astonishment. This means that a semantically equivalent response, even in cases where the resource has been deleted, is the least surprising approach that still delivers a consistent experience when using tools like the AWS SDK and AWS CLI.

Building on our earlier EC2 instance launch example, we can model this edge case by terminating the EC2 instance that we previously launched:

```
$ aws ec2 terminate-instances --instance-ids i-xxxxxxxxxxxxxxxxx
{
  "TerminatingInstances": [
    {
```

```
            "InstanceId": "i-xxxxxxxxxxxxxxxxx",
            "CurrentState": {
                "Code": 32,
                "Name": "shutting-down"
            },
            "PreviousState": {
                "Code": 16,
                "Name": "running"
            }
        }
    ]
}
```

If we follow this with a simulated "late arriving" retry of the RunInstances API request with the same parameters and unique request identifier, we will get a semantically equivalent response. However, as the following example shows, the instance is now terminated:

```
$ aws ec2 run-instances --image-id ami-04fcd96153cb57194 --instance-type t2.micro --client-token eb3c3141-a229-4ca0-b005-eb922e2cabdc
{
    "Instances": [
        {
            "Monitoring": {
                "State": "disabled"
            },
            "StateReason": {
                "Message": "terminated", ← Now terminated
                "Code": "terminated"  ← Now terminated
            },
            "State": {
                "Code": 48,  ← Terminated status code
                "Name": "terminated"  ← Now terminated
            },
            "InstanceId": "i-xxxxxxxxxxxxxxxxx",
            "ImageId": "ami-04fcd96153cb57194",
            …
            "ClientToken": "eb3c3141-a229-4ca0-b005-eb922e2cabdc",
            …
```

To support this behavior we need to retain knowledge of the initial idempotent request in our service. However, it's impractical to retain that knowledge indefinitely. Indefinite retention could also have an undesirable impact on the customer experience if some future request identifier collides with one used much earlier. Requirements for retention vary across services and service resources. We have found that, for EC2 instances, it works to limit the time period to the lifetime of the resource, plus an interval after which it is reasonable to assume that any late arriving requests would either have arrived or would no longer be valid.

Responding when a customer changes request parameters on a subsequent call where the client request ID stays the same

We design our APIs to allow our customers to explicitly state their intent. Take the situation where we receive a unique client request token that we have seen before, but there is a parameter combination that is different from the earlier request. We find that it is safest to assume that the customer intended a different outcome, and that this might not be the same request. In response to this situation, we return a validation error indicating a parameter mismatch between idempotent requests. To support this deep validation, we also store the parameters used to make the initial request along with the client request identifier.

Conclusion

Werner Vogels, Amazon CTO, notes that one of the lessons we have learned at Amazon is to expect the unexpected. He reminds us that failures are a given, and as a consequence it's desirable to build systems that embrace failure as a natural occurrence. Coding around these failures is important, but undifferentiated, work that improves the integrity of the solution being delivered. However, it takes time away from investing in differentiating code.

In the article, "Timeouts, retries, and backoff with jitter," discussed earlier, Marc Brooker shows how we can leverage retries to mitigate transient faults, and how automating these retries in SDKs and tooling can improve usability. Building on that article, in this article we explore how AWS leverages idempotent behavior to deliver the benefits of automated retry policies to our customers while still honoring the "at most once" commitment inherent in the API contracts of some operations.

We have found the approach we've outlined works well, and we have heard that customers appreciate the reduced complexity this approach delivers. However, there is cost and complexity inherent in building services to meet the contract this article describes, and that complexity is not right for all solutions. Sometimes it's better to take the extra time to meet the demands of a more rigorous contract as described here, in other cases it's better to innovate more quickly with a less complex contract because that's best for our customers. In cases where a more rigorous "at most once" contract is required, we have found the approach in this article to work well to meet our customers' needs.