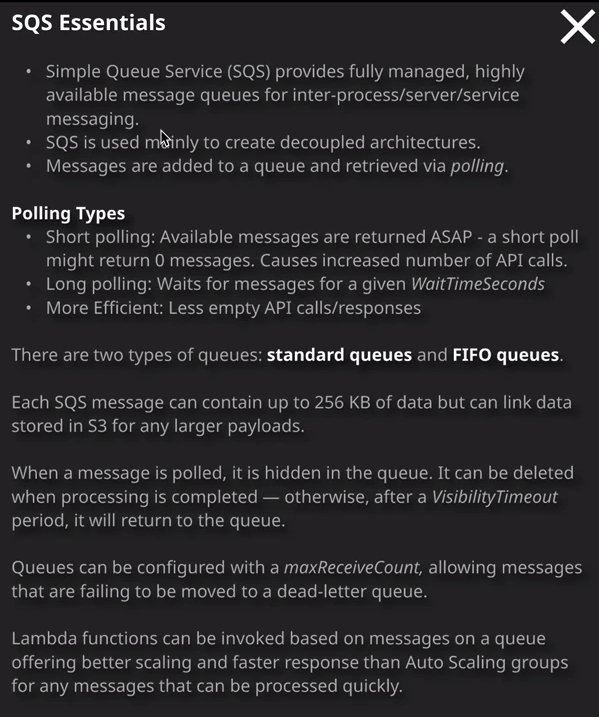
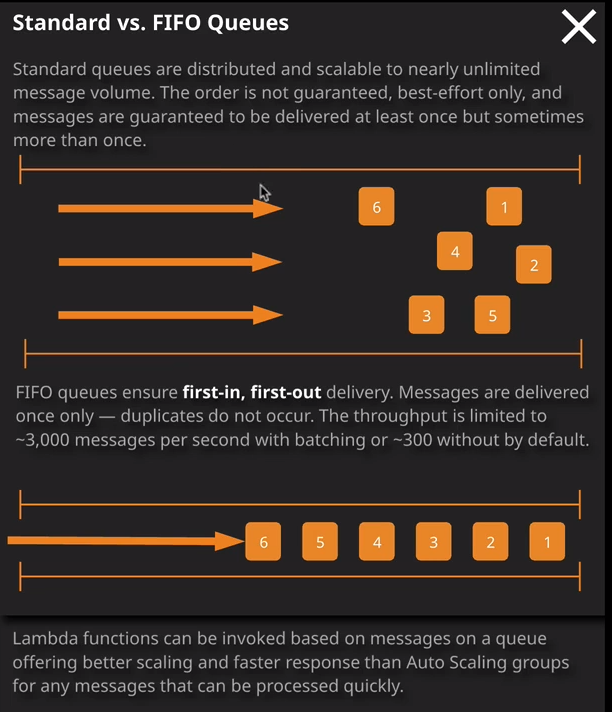
Welcome back and in this next lesson, I want to talk about SQS or the simple queue service.

So **simple queue service is a product that tends to feature in many of the same architectures as the simple notification service.** Whilst it is a separate product in its own right, it is often complimentary to SNS. So you often find SNS and SQS working together to improve the scalability and performance of a lot of architectures. So in this lesson, I want to focus on the key parts of SQS as they're relevant for the exam. So before I get started talking about the architecture of the benefits of SQS I want to cover some of the basics



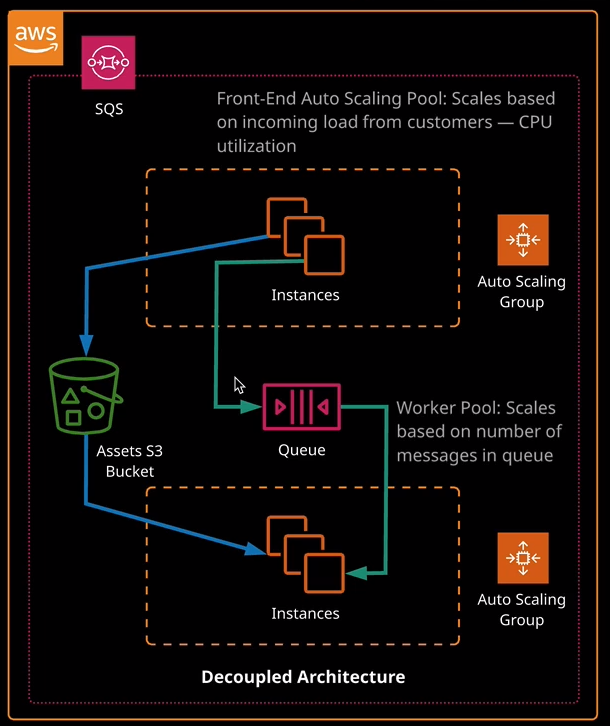
so **SQS is essentially a queuing system, it provides fully managed, highly available message queues and these message queues could be used for interprocess, interserver, or interservice messaging, so the concept of a queue is pretty simple. You have a queue. You add a message to the queue and then something else can retrieve messages from that queue. So it offers you an asynchronous way to talk between two components of an application. You can take an item, you can add it onto the queue, and something else can take the item off the queue and do something with it, so queues are often used for worker pool type architectures**. I want you to think of a website like YouTube, where people are uploading videos to that website and they need to have some form of processing performed on them. Well, maybe the frontend allows the user to upload the video stores that video on S3 and then adds an item to the queue indicating that some processing is required and then, at the backend, you might have a fleet of EC2 instances that are constantly checking that queue for messages and when a message arrives, it takes it off the queue, finds the media file that it corresponds to and then performs the processing. So queues can be used in that form of worker pool messaging architecture but I'll talk more about that later in the lesson.

Now, **SQS messages that you add to a queue can contain up to 256 kilobytes of data, so that maximum size is exactly the same as messages that could be sent to an SNS topic and that's by design, because again, these two products often work in cooperation. So you want the maximum limit of data to be the same with both of the products**.   
Now, you are doing any sort of jobs where you need to process any larger payloads than 256 kilobytes. Then you can always store the main data in S3 and have the message that gets added to the queue just containing a link to that bigger file. So when messages are on the queue, in order to get them off the queue, you need to follow a process called **polling**. Now polling can take two different forms. You can do what's called **short polling** and short polling is a single API call where you check the queue for any messages. If there are any messages on the queue, then you get those messages delivered, and you can define how many messages you want to retrieve anywhere from a single message all the way through to a maximum of 10. Now, the key thing about short polling, is you're going to get one of two responses, you're either going to get the messages that are available on the queue at that point in time, up to a maximum of 10 or if there's nothing on the queue, you're going to get an immediate response of no messages. Now, if you want to able to respond immediately when a message arrives on a queue using ***short polling, you're going to have to do a constant set of API calls***. You need to check if the queues got any messages, then do it again and again. It's almost like a loop and you're constantly doing it, consuming API calls and that's relatively inefficient. Now the alternative is to use what's known as **long polling and long polling is a process where you do the same check for messages on the queue but you wait for a certain amount of time, and that time is known as the wait time seconds**. So you can specify a time that you're willing to wait until that queue has any messages, and whenever messages arrive on the queue during that wait time seconds, you'll get them immediately delivered to you so you don't have to keep polling the queue for new messages. You could just initiate a long polling request and wait for it to get messages. Now that's much more efficient because you have a lot less empty API calls. You have a lot less calls to the queue that receive no messages. Now, the way that message received works, I want to imagine that you're an EC2 instance and you doing either a short poll or a long poll for messages on a queue. Well, you'll see a list of messages, and you'll receive those messages and straight away those messages will be hidden from the queue. They won't be deleted. They'll be just hidden and they're hidden for an amount of time known as the **visibility timeout**. Now, when that visibility timeout expires, let's say that it's 30 seconds. Those messages will return onto the queue, and any other polling operations will receive a copy of that message again. What you need to do is an EC2 instance in this example is when you poll the queue and receive the message. Once you've performed the processing, you need to go back and delete the message, and once you've deleted it, then it's gone from the queue forever. Now that's a really important architecture and that's how **queues offer automatic retry and automatic high availability because when they're being polled, if you don't acknowledge that you've dealt with the message, it becomes visible again for another retry or another process**. So that's important to know for the exam.



Now SQS queues come in two different types. **We've got standard queues** which were the original SQS type available in AWS and we've more recently gotten **FIFO queues which stands for first in first out.** Now there is some crucial differences between these type of queues that you need to be aware of for the exam. **So a standard queue is capable of a near unlimited throughput, so standard SQS queues are distributed. They're operating on lots of different underlying pieces of hardware in AWS at the same time.** So I want you to imagine a standard SQS queue as a fairly wide highway, so you've got lots of messages flowing down this highway. There's lots of capacity, but the problem is when you're polling the queue so you'll be on this side, the right side of the queue. You could receive the messages out of order. You might not get those messages in the same order as you intend. So this is crucial. So standard queues offer near unlimited throughput. They offer at least once delivery. **So it is possible in extreme circumstances that you might have duplicate messages on the queue.** So that's important to understand. If you're designing solutions, you need to assume that you could, in certain circumstances, receive more than one copy of the message. **The only thing that standard queues guarantee is that you will get at least one copy of that message and standard queues also only offer best effort ordering, and that means that occasionally when you're receiving messages at the destination end of the queue, they might not always be in the same order that they've been added in** and as a solutions architect, you need to design any solutions that you implement using standard queues with that in mind. **So they need to be able to tolerate some disordering of messages.** **So with standard queues you're essentially getting really good performance, really good scalability and resilience but before that you need to accept some tradeoffs. You could get message deliveries more than once, and in certain circumstances they could be out of order**.

Now these problems are addressed by **FIFO queues which are known as first in first out and as the name suggests, they guarantee that messages are delivered once and once only**. So you do not get any duplicates occurring with a FIFO queue and you're also guaranteed that the order that you add messages to that queue will be the order that message is received in. Now **the trade off for that is throughput is limited to 3000 messages per second with matching or 300 without**. So you get a lot **lower levels of performance with FIFO queues than you do with standard.**



So queues are often used in situations where you need to decouple to different parts of the system. So the example that's normally used to illustrate the power of queues is an example of a website like YouTube. So I want you to imagine this website is constructed of two different sets of EC2 instances. You've got a frontend tier of EC2 instances and these are using an auto scaling group. Let's imagine they've got a load balancer and an auto scaling group, and these are fully scaleable to cope with whatever demand is placed on them from your customers. So what these incidences do is they accept the uploads of these huge 4K media files and they store them in an S3 bucket and at the same time, they add a message to an SQS queue and that message just indicates that the needs to be some processing occurring on those 4K videos. At the other side of the queue, we have a worker pool and again, this is in an auto scaling group, so we can have one instance or we could have 100 instances or perhaps even more. Essentially, the auto scaling group of this worker pool is scaling based on the number of messages in this queue. So this worker poll keeps polling this queue, receives a message, does some media transcoding, and then puts the result in asset back onto S3.   
Now what this architecture means is **because the size of this worker pool is based on the number of messages in this queue, the more video uploads you get, the more messages in the queue and the more instances inside this worker pool so it auto scales based on demand and overtime this reaches an equilibrium where you're always getting video processing occurring in a timely way** but what this architecture also means is that the **frontend and backend are decoupled from each other**. It means they can work independently. So the uploads can happen to the frontend istances and the processing can happen on the backend. Each of these components can scale independently and they can fail independently. I want you to imagine you have a major failure and all of these backend work pool instances are out of action, we'll still be able to provide the service to our customers because these instances will still be operational, they can still accept uploads. Those uploads can still be added to S3, which is a highly resilient service. Messages can still be added to the queue and as far as our customers are concerned processing is occurring in the background. Our administrators might identify what the problem is fix these instances and immediately bring them back into action and at that point the auto scaling group will begin scaling based on the messages in these queues and so you've decoupled the frontend from the backend. You've allowed **one to fail without affecting the other. You've allowed one to scale without the other also having to scale. So if the CPU load caused by these video uploads is fairly low, then you might only have two or three instances but if these instances are adding, say, tens or hundreds or thousands of messages to this queue then this backend pool of worker instances could be significantly larger**.

So if you face any exam questions which talk about asynchronous messaging or decoupling of different application components or the ability to have independent scaling of different components of your application, then you should by default, think about **SQS** queues as the best answer.