

**Characteristics of distributed systems**

we're focusing specifically on web applications.

There's going to be a lot of different architecture styles, so you can still have a client-server sort of architecture where you've got a client talking to a server for info and presenting it back to the user and clients sending data back to the data for persistence, things like that.

You can get to these three tiers or end tier architectures where the client logic moves to a middle layer, so clients are stateless. End tier, you may have again additional layers if there are things like caching or other app services.

You can have a cluster style, where clusters of machines work together.

A peer-to-peer network where there's no single controller, but all these machines have some sort of equal responsibility and you can do a space based sort of thing where you're using SOA, and REST, of an adventure of an architecture, where you have this distributed machines that all scale independently, and have shared nothing. So a lot of different architecture styles.

-🡪**Data sharing** is going to be different. You might have a :  
**shared databases** where everyone's dependent on a single data source.

**Synchronization** is going to come into play if you're having different replicas of data of some sort of no single master shared nothing, but where data has to be synchronized among multiple nodes and so that's where you're going to look at **replication** of **messaging**, or other different technologies and ways to keep things in sync.

**Caching** is going to come into play as you have data sharing, is how can I cache information that might be slow changing to make sure performance is strong when I am expanding geographies and dealing with latency.

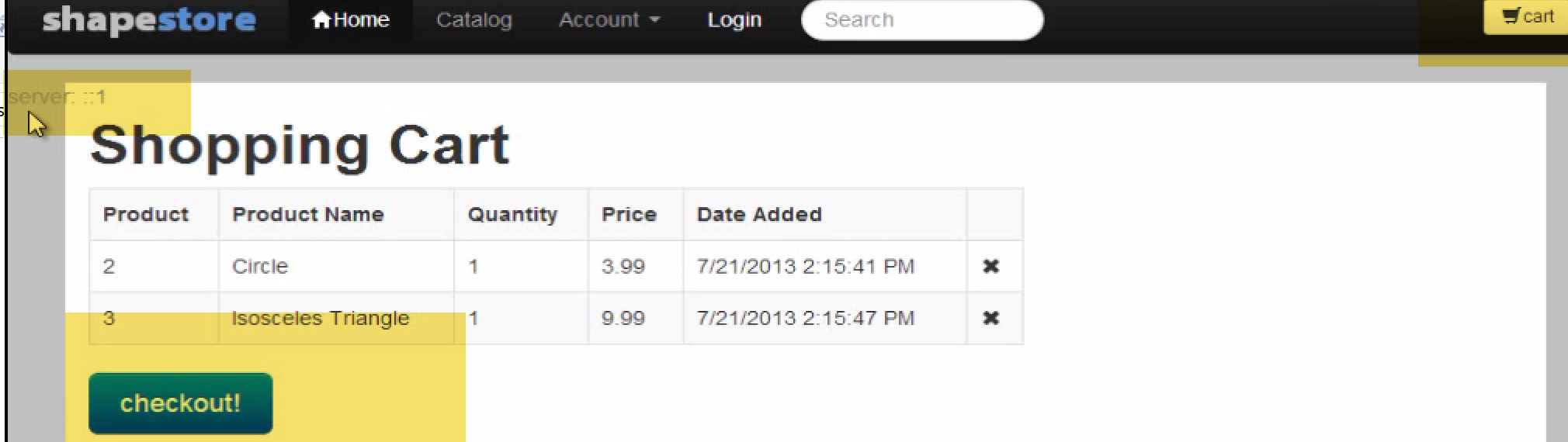
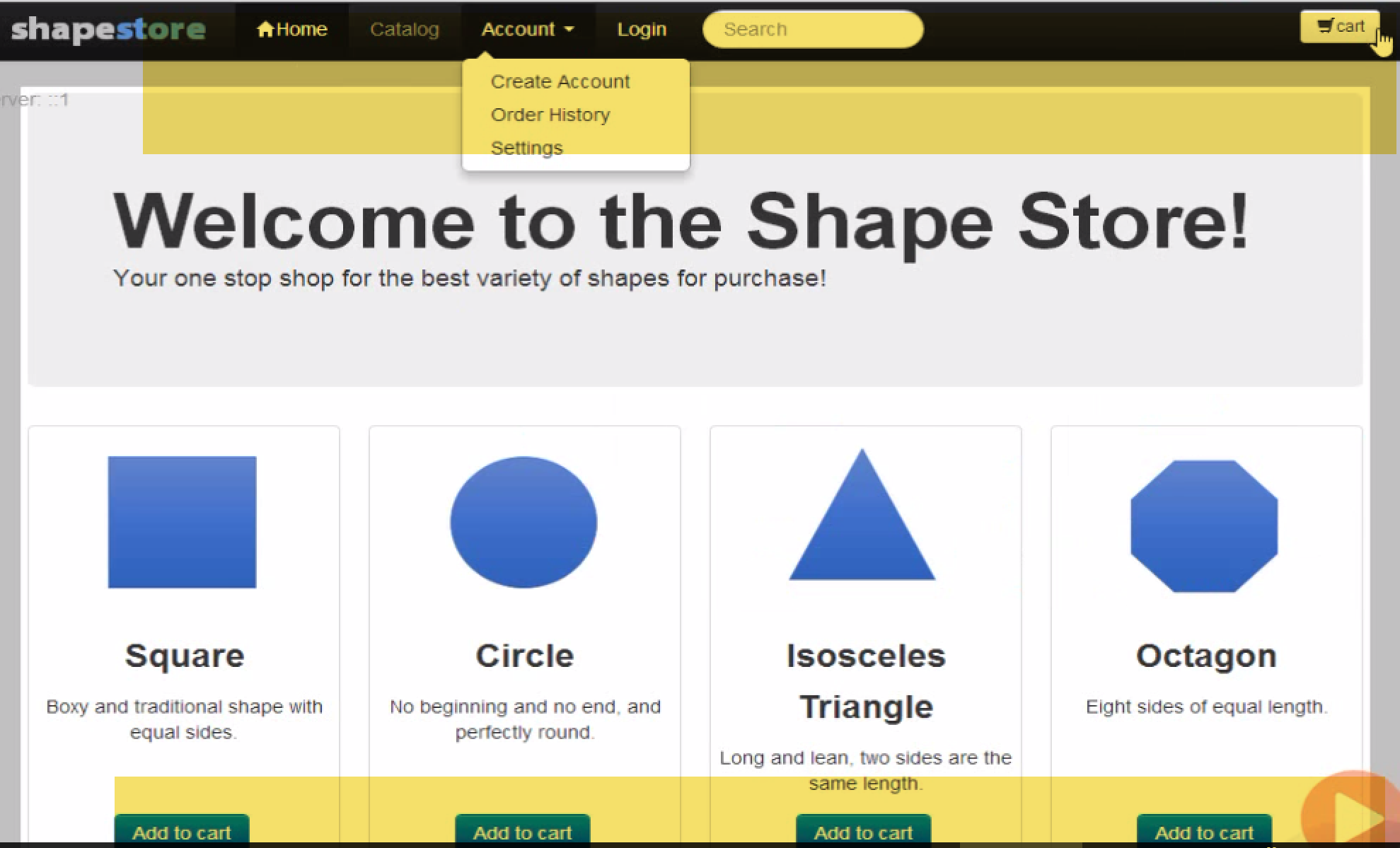
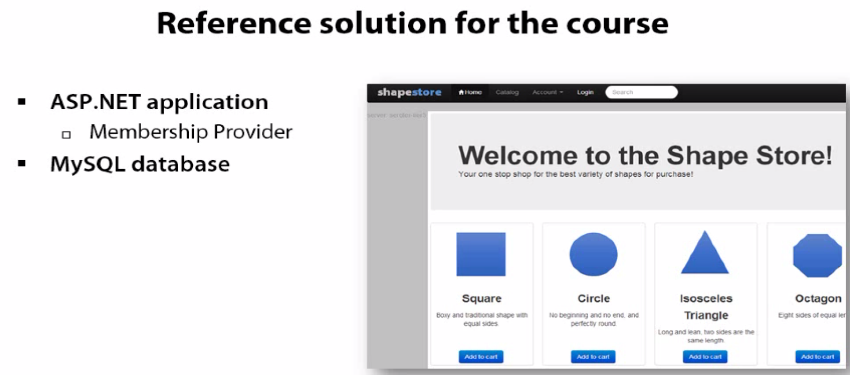
A lot of different **fault tolerant techniques** when you're dealing with distributed web systems. Whether you're working with timeouts and retires to bake in, to make sure you don't have worry about if a single thing fails, can you try it again. Having isolation, things like separate threads, so that again a single failure doesn't cascade and that's where you get into things like circuit breakers, where how do I make sure that I don't have cascading errors and if something fails, I can even switch over to a stub or something else that might be using caches, or no responses, or even default data to handle a failure of a data subsystem or even just use eventual consistency and drop a message into a queue and it'll synchronize later. I'm going to use fail fast techniques in these distributed systems where again, latency is often worse than something failing completely. So you want things if they're not performing, fail and move on and there's going to be a lot of these

core principles like **availability**. That's going to be something you're going to focus on a distributed web system you have to architecture for. You can't assume it's baked in on any platform especially the Cloud. **Reliability**, again this thing has to be consistently working when you're distributing over this, you have to think about that in your architecture.

How can you **scale**? That means how can I continue to increase and decrease load and have the environment grow and shrink as necessary without reducing my functionality?

**Performance**, can I get consistent performance? Can I handle spikes in performance? Can I make sure that regardless of what I'm sending at the system, it's going to perform okay?

And then **manageability**. How can I make sure that even if I'm expanding all of these different layers and subsystems and owners? I'm not dealing with a management nightmare where I can't ever trace an actual problem….

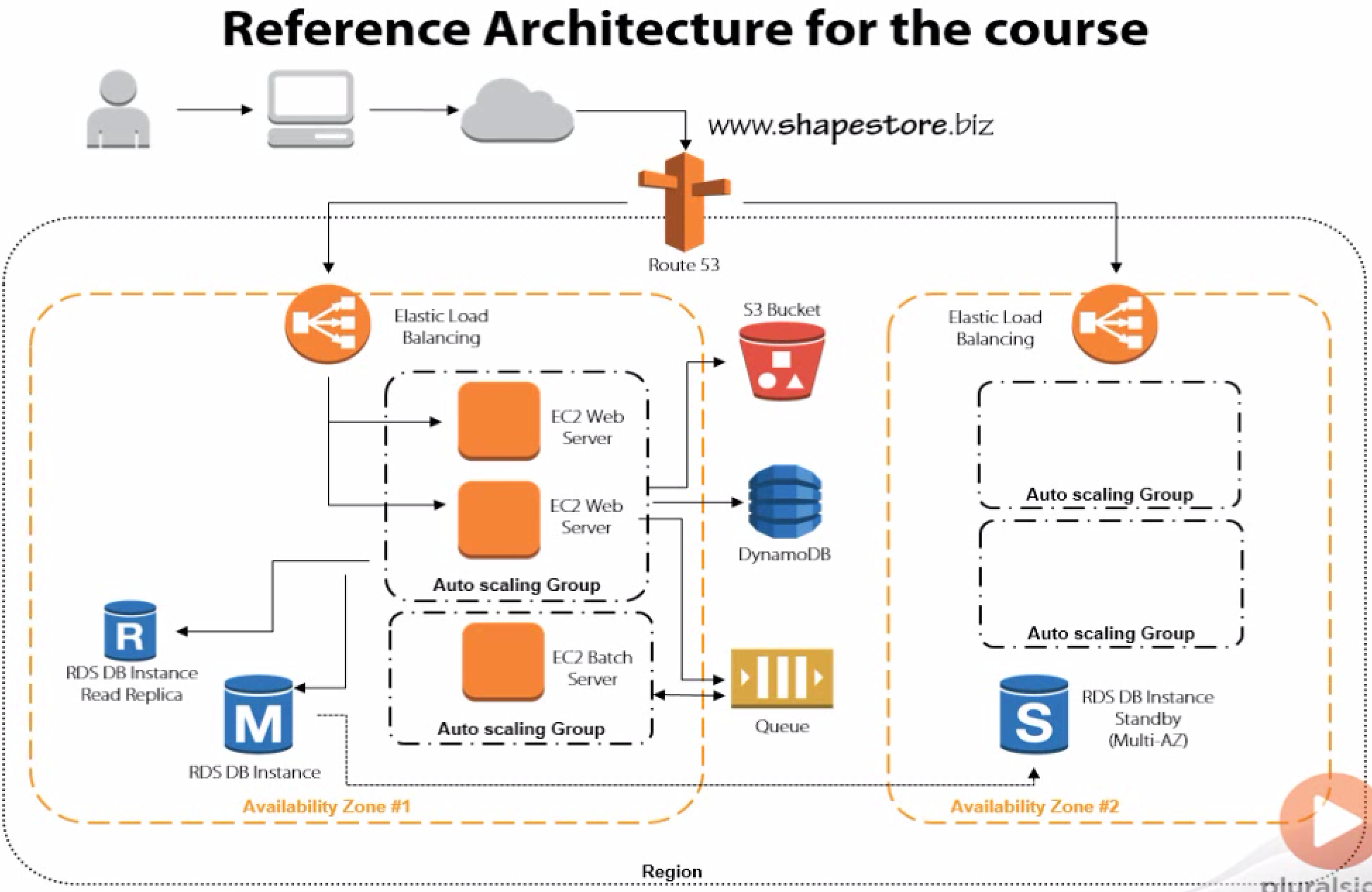


**Reference architecture for this course**

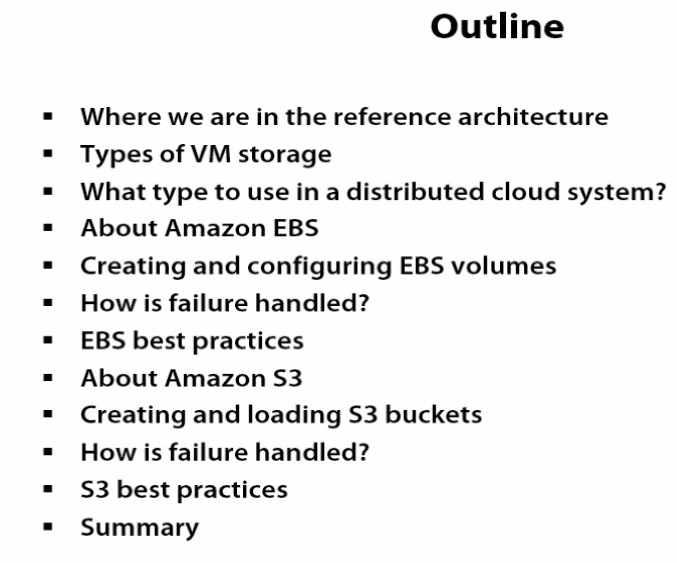
Web traffic that comes into shapestore.bix, I went ahead and registered that domain. That data is going to come into an Elastic Load Balancer, which is going to route traffic to one of many different servers, those are all part of a scaling group so they'll scale as needed.

Static Content is stored in Amazon S3,

Session State being stored in DynamoDB, so it can expand servers.

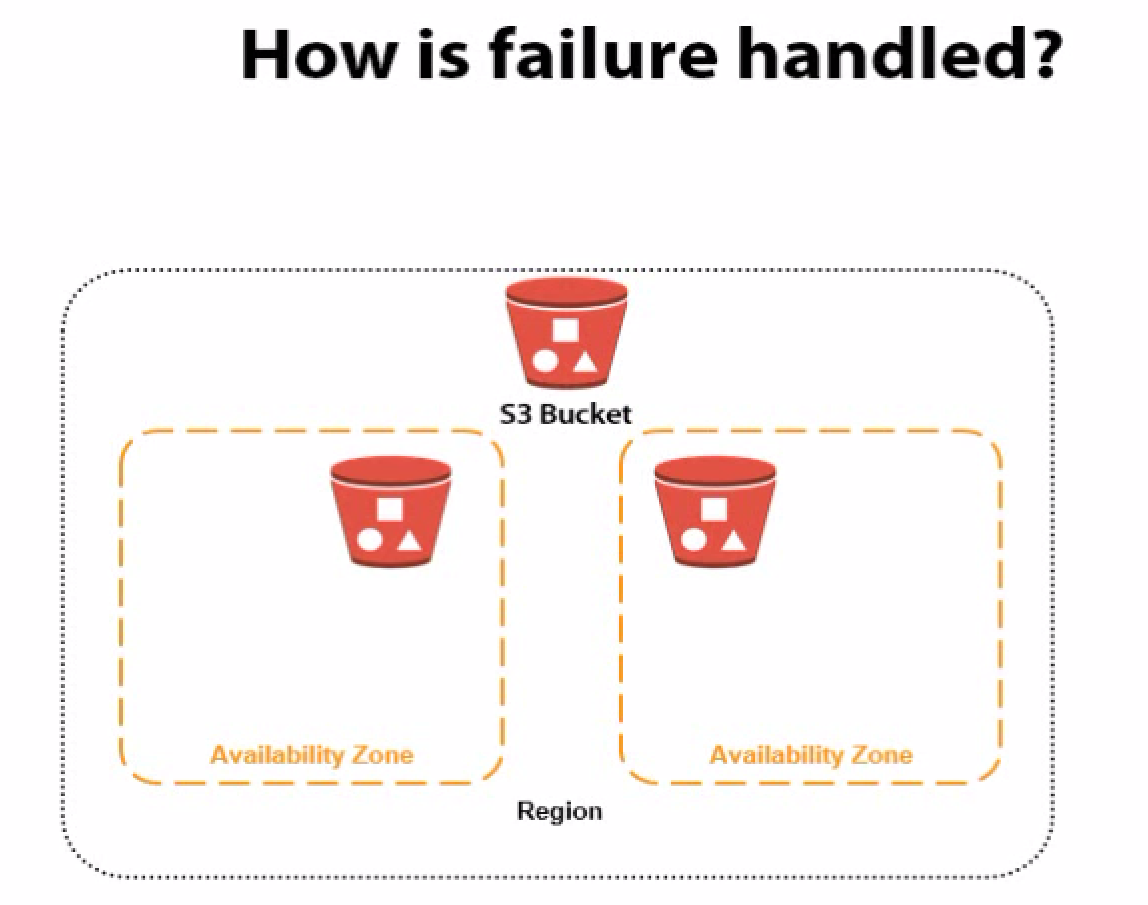
I'm storing data in RDS, the Relational Database Store for things like the product catalog and I'm getting the orders in there. But of course, orders are first going into a queue and I have a separate set of servers that are going to read the orders from the queue and process them, so that if I have a huge spike, I'm not putting a constraint on the database directly, I'm just going to keep filling up the queue and I'm going to have servers that process those kind of asynchronously, so I reduce my coupling. That relational database is automatically replicated to a different availability zone, so I have a standby server. So I could lose an entire availability zone and everything would keep running.

**Provisioning Durable Storage with EBS and S3:**

****

**How is failure handled?**

Handling failure with S3 is pretty straightforward mainly because everything's geo redundant by default. So even if I lose an entire Availability Zone, everything really just keeps working because you've got that redundancy, the address doesn't change, so when I'm accessing that bucket URL that data's while spread across AZs I'm really funneling it through one address. So even if I lose an Availability Zone, nothing really bad happens and as you saw as well, another form of failure is having a poor performing website, I can also use S3 to host a static HTML website if I want to point to a, let's say a less busy static highly available site, in the case of a degraded performance.

****

**S3 best practices**

