Design Amazon

Design the system that powers Amazon's e-commerce business.

*Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.*

*We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.*

Clarifying Questions To Ask

**Question 1**

***Q: Amazon's e-commerce business has a lot of functionality, but at its core, it involves going on the Amazon home page, looking at recommended items, searching for items, browsing through lists of items and individual items, adding them to carts, submitting orders, and maybe even cancelling orders. Is this the user flow that we're designing a system for?***

A: Yes, but for the sake of this design, let's keep things simple and assume that the Amazon home page consists of just a search box. In other words, you don't have to worry about listing recommended items on the home page. Also, you can assume that users can only have one cart at a time; they can't save multiple carts.

**Question 2**

***Q: Should we design the subsystem / recommendation engine that determines what to return when a user searches for items?***

A: You can assume that there's a service that returns relevant items given some search parameters that are taken in through the Amazon search box. These services alone can get pretty complex, so let's focus on designing the rest of the system for this question. But you can of course use this service as part of your design.

**Question 3**

***Q: Should we handle what happens when an item is out of stock, or are we assuming that items are always in stock for this question?***

A: Yes, you should handle this. Specifically, you should prevent users from adding items that are out of stock to their cart, and orders should naturally "consume" stock. This whole item-stock issue is a pretty important part of the e-commerce system that you should focus on.

**Question 4**

***Q: How should we handle items that have low stock and are being viewed by multiple people at the same time? In other words, should we "reserve" items in some sense?***

A: This can be a pretty complex part of the system, depending on how fancy you want to be. Let's keep things fairly simple and design this as follows: if an item, on its view page, is in stock, any user can add it to their cart, and adding it to their cart doesn't "consume" stock. Once a user begins the checkout process, the system should alert them if an item in their cart has gone out of stock since they added it, and otherwise, it should effectively "reserve" the item during the duration of the checkout process, capped at, say, 10 minutes.

**Question 5**

***Q: Should we design the part of the system that handles what happens after an order is submitted? For example, orders are dispatched to Amazon warehouses, workers are assigned to packages, etc..***

A: Amazon orders often consist of items that are scattered across multiple Amazon warehouses. This means that some orders end up being split up into multiple suborders, each assigned to relevant warehouses. You should think about how relevant warehouses will be notified of orders or suborders and how the stock of items within warehouses might come into play. But you don't have to worry about what happens after a warehouse has been assigned an order or suborder.

**Question 6**

***Q: In the event that a single order or suborder can be handled by multiple Amazon warehouses, how should our system figure out which warehouse to route that order to?***

A: Similar to the recommendation engine that we touched on earlier, you can assume that we have access to some smart service that handles the logic of assigning orders to warehouses. Don't worry about the service itself—just how it interacts with other parts of the system. But again, you should think about how item stock in individual warehouses will come into play here.

**Question 7**

***Q: Do we want to handle auxiliary Amazon features, like Amazon Prime, subscription purchases, purchasing items as used vs. new, etc.?***

A: No.

**Question 8**

***Q: Amazon has a lot of regional websites / stores. For example, amazon.com, amazon.fr, amazon.in. Should we design all of these, or can we design just amazon.com, for example, and then effectively assume that the design will work for all of the other major regional Amazon websites?***

A: For the sake of this design, let's just design amazon.com, and you can assume that amazon.com and other Amazon websites are entirely separate, yet identical businesses / systems. So whatever you design for amazon.com will be applicable to amazon.fr, amazon.in, and other Amazon websites.

**Question 9**

***Q: As far as latency and reliability are concerned, I'm assuming that we want Amazon to be mostly highly available, but is it ok if there's a little bit of loading time when you search for items, when you submit an order, etc.?***

A: Ideally, searching for items should have low latency, but it's fine if beginning the checkout process and submitting orders takes a bit of time. As far as reliability is concerned, let's not worry too much about it for the sake of this design. You can assume that you have a highly available SQL setup out of the box, without worrying about the availability details. I'm more interested in the database schemas that you're going to use to support the core functionality.

**Question 10**

***Q: Amazon e-commerce is obviously a very large-scale system. How many customers are we dealing with, and how many orders can we expect per day?***

A: Amazon has roughly 300 million customers and processes about 60 thousand orders per hour, which means roughly 20 orders per second. If you're designing this specifically for amazon.com, let's assume that the U.S. constitutes 50% of all Amazon orders globally. So roughly 30 thousand orders per hour or 10 orders per second. That being said, once again, for this design, I'm more interested in how your system will support the core functionality of Amazon's e-commerce business rather than its scale.

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

**1. Gathering System Requirements**

As with any systems design interview question, the first thing that we want to do is to gather system requirements; we need to figure out what system we're building exactly.

We're designing the e-commerce side of the Amazon website, and more specifically, the system that supports users searching for items on the Amazon home page, adding items to cart, submitting orders, and those orders being assigned to relevant Amazon warehouses for shipment.

We need to handle items going out of stock, and we've been given some guidelines for a simple "stock-reservation" system when users begin the checkout process.

We have access to two smart services: one that handles user search queries and one that handles warehouse order assignment. It's our job to figure out how these services fit into our larger design.

We'll specifically be designing the system that supports amazon.com (i.e., Amazon's U.S. operations), and we'll assume that this system can be replicated for other regional Amazon stores. For the rest of this walkthrough, whenever we refer to "Amazon," we'll be referring specifically to Amazon's U.S. store.

While the system should have low latency when searching for items and high availability in general, serving roughly 10 orders per second in the U.S., we've been told to focus mostly on core functionality.

**2. Coming Up With A Plan**

We'll tackle this system by first looking at a high-level overview of how it'll be set up, then diving into its storage components, and finally looking at how the core functionality comes to life. We can divide the core functionality into two main sections:

* The user side.
* The warehouse side.

We can further divide the user side as follows:

* Browsing items given a search term.
* Modifying the cart.
* Beginning the checkout process.
* Submitting and canceliing orders.

**3. High-Level System Overview**

Within a region, user and warehouse requests will get round-robin-load-balanced to respective sets of API servers, and data will be written to and read from a SQL database for that region.

We'll go with a SQL database because all of the data that we'll be dealing with (items, carts, orders, etc.) is, by nature, structured and lends itself well to a relational model.

**4. SQL Tables**

We'll have six SQL tables to support our entire system's storage needs.

**Items**

This table will store all of the items on Amazon, with each row representing an item.

| **itemId: *uuid*** | **name: *string*** | **description: *string*** | **price: *integer*** | **currency: *enum*** | **other...** |
| --- | --- | --- | --- | --- | --- |
| ... | ... | ... | ... | ... | ... |

**Carts**

This table will store all of the carts on Amazon, with each row representing a cart. We've been told that each user can only have a single cart at once.

| **cartId: *uuid*** | **customerId: *uuid*** | **items: *[]{itemId, quantity}*** |
| --- | --- | --- |
| ... | ... | ... |

**Orders**

This table will store all of the orders on Amazon, with each row representing an order.

| **orderId: *uuid*** | **customerId: *uuid*** | **orderStatus: *enum*** | **items: *[]{itemId, quantity}*** | **price: *integer*** | **paymentInfo: *PaymentInfo*** | **shippingAddress: *string*** | **timestamp: *datetime*** | **other...** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

**Aggregated Stock**

This table will store all of the item stocks on Amazon that are relevant to users, with each row representing an item. See the **Core User Functionality** section for more details.

| **itemId: *uuid*** | **stock: *integer*** |
| --- | --- |
| ... | ... |

**Warehouse Orders**

This table will store all of the orders that Amazon warehouses get, with each row representing a warehouse order. Warehouse orders are either entire normal Amazon orders or subsets of normal Amazon orders.

| **warehouseOrderId: *uuid*** | **parentOrderId: *uuid*** | **warehouseId: *uuid*** | **orderStatus: *enum*** | **items: *[]{itemId, quantity}*** | **shippingAddress: *string*** |
| --- | --- | --- | --- | --- | --- |
| ... | ... | ... | ... | ... | ... |

**Warehouse Stock**

This table will store all of the item stocks in Amazon warehouses, with each row representing an {item, warehouse} pairing. The *physicalStock* field represents an item's actual physical stock in the warehouse in question, serving as a source of truth, while the *availableStock* field represents an item's effective available stock in the relevant warehouse; this stock gets decreased when orders are assigned to warehouses. See the **Core Warehouse Functionality** section for more details.

| **itemId: *uuid*** | **warehouseId: *uuid*** | **physicalStock: *integer*** | **availableStock: *integer*** |
| --- | --- | --- | --- |
| ... | ... | ... | ... |

**5. Core User Functionality**

**GetItemCatalog(search)**

This is the endpoint that users call when they're searching for items. The request is routed by API servers to the smart search-results service, which interacts directly with the **items** table, caches popular item searches, and returns the results.

The API servers also fetch the relevant item stocks from the **aggregated\_stock** table.

**UpdateCartItemQuantity(itemId, quantity)**

This is the endpoint that users call when they're adding or removing items from their cart. The request writes directly to the **carts** table, and users can only call this endpoint when an item has enough stock in the **aggregated\_stock** table.

**BeginCheckout() & CancelCheckout()**

These are the endpoints that users call when they're beginning the checkout process and cancelling it. The *BeginCheckout* request triggers another read of the **aggregated\_stock** table for the relevant items. If some of the items in the cart don't have enough stock anymore, the UI alerts the users accordingly. For items that do have enough stock, the API servers write to the **aggregated\_stock** table and decrease the relevant stocks accordingly, effectively "reserving" the items during the duration of the checkout. The *CancelCheckout* request, which also gets automatically called after 10 minutes of being in the checkout process, writes to the **aggregated\_stock** table and increases the relevant stocks accordingly, thereby "unreserving" the items. Note that all of the writes to the **aggregated\_stock** are ACID transactions, which allows us to comfortably rely on this SQL table as far as stock correctness is concerned.

**SubmitOrder(), CancelOrder(), & GetMyOrders()**

These are the endpoints that users call when they're submitting and cancelling orders. Both the *SubmitOrder* and *CancelOrder* requests write to the **orders** table, and *CancelOrder* also writes to the **aggregated\_stock** table, increasing the relevant stocks accordingly (*SubmitOrder* doesn't need to because the checkout process already has). *GetMyOrders* simply reads from the **orders** table. Note that an order can only be cancelled if it hasn't yet been shipped, which is knowable from the *orderStatus* field.

**6. Core Warehouse Functionality**

On the warehouse side of things, we'll have the smart order-assignment service read from the **orders** table, figure out the best way to split orders up and assign them to warehouses based on shipping addresses, item stocks, and other data points, and write the final warehouse orders to the **warehouse\_orders** table.

In order to know which warehouses have what items and how many, the order-assignment service will rely on the *availableStock* of relevant items in the **warehouse\_stock** table. When the service assigns an order to a warehouse, it decreases the *availableStock* of the relevant items for the warehouse in question in the **warehouse\_stock** table. These *availableStock* values are re-increased by the relevant warehouse if its order ends up being cancelled.

When warehouses get new item stock, lose item stock for whatever reason, or physically ship their assigned orders, they'll update the relevant *physicalStock* values in the **warehouse\_stock** table. If they get new item stock or lose item stock, they'll also write to the **aggregated\_stock** table (they don't need to do this when shipping assigned orders, since the **aggregated\_stock** table already gets updated by the checkout process on the user side of things).

