## Designing an API

### Course Overview

Have you ever wondered how data makes it from your favorite social network to your mobile device and back? How about that email from your computer to your colleague's inbox? I'm Markus Neuhoff, and I'm excited to welcome you to the world of exchanging data using REST. Not only will we answer these questions and more, but we'll build our very own REST API too. REST, like many protocols, doesn't have specific versioning. In fact, there haven't been any changes to the protocol since it was created in the year 2000. However, I am using the displayed versions of Node and its web framework Express to explain how REST works in this course. And this slide shows the versions of Node and Express for which the information in this course applies. Now before we get too much further, you might be wondering, is learning REST really worth it? And I would resoundingly say yes. In fact, most if not all devices that are capable of making an internet connection have the ability to consume or interact with REST APIs. But how exactly do they talk to each other? We'll tackle that next.

### REST URLs and Request Verbs

While REST is an incredibly flexible protocol, there are definitely some conventions that allow us all to speak a common language. Let's say you're building an API that manages the list of items. One of the first things you need to do is decide how consumers like a mobile app will access your data. But more simply, you need to give your resource, the list of items, a name. We'll be using Carved Rock Fitness, a fictional outdoor equipment supplier, throughout this course, so let's take a look at one of their URLs. First, we have the scheme, which for REST is most commonly HTTPS for secure connections and HTTP for insecure ones. Unless you're actively working in your local environment, it is always a good idea to use HTTPS. Next, we have the domain or the name of the server that's hosting your service. Seeing a number behind your domain may be a new experience for you. This number represents the port or specific part of the server that will be responding to requests. 443 is the default port for HTTPS, and 80 is the default for HTTP. Anytime you're serving content on these default ports, you can emit them. In fact, the only reason that I mentioned it at all is that the local development port typically takes place on ports other than 80 or 443, so that's why I included it here. So we're on the right server. Now what? The path tells us specifically where the resource that we want to access is. More importantly, we're asking for the API portion of the server and then a particular item with the ID of 1. And finally, we can also include additional information in our request through the query parameters. In this case, we're asking that the item's data include any images of the item. Let's dig in the path a little bit more since if we're being honest, they're carrying most of the weight in the API naming convention. In the example we just talked about, I'm starting the path with api. Now this is optional, but there are two main reasons to do this. First, it improves readability of the traffic, which is great for debugging purposes. And second, some organizations host both a web front end and an API on the same servers, so the distinction helps keep things separate. Now over time, APIs inevitably change. And since you're creating a contract between your API and your consumers, you'll want to create a versioning system to highlight breaking changes. We've got a whole module on versioning coming up in a bit. But for now, just know that adding version information in the path is a common approach. While referring to specific groups of resources like items or customers or orders, it's best practice to refer to them in the noun version and to keep it plural. For example, we would want to just use items instead of get items or list items. And if we wanted the collection instead of resources, we would just stop there and we wouldn't pass any additional path values. This pattern communicates to our consumers that they can expect one or more items to be returned. And finally, if we specify an identifier like an integer or a UUID or a unique name, we can expect to get that specific item back. Another common REST pattern is subcollections. For example, let's imagine that we have an endpoint that gives us a specific order. We maybe want to get data for this order, but just the generic order information for now like the customer status, that sort of thing. However, we can create subcollections to return all of the items in that order too. And leveraging the same pattern of following a collection with an identifier, we could get a specific item by its ID. The final part of the URL is the optional query, which broadly allows you to modify your requests. Let's talk through some common permutations and use cases. As we saw in the path example earlier, it's very easy to include a single query with a single value, for example, letting the server know that we want to include the images for our items. It's also perfectly acceptable to include multiple values within a query separating them by commas. Here, I'm letting the server know that I would like both images and reviews of the item that I'm requesting. Now most languages will take this comma‑separated list and automatically convert it into an array or similar collection for you. Another common use case for queries is paging. Here I'm saying that I would like 50 results, starting with the 51st record. This is the URL I would call if I was requesting the second page of data with 50 items per page. Notice that the queries are separated by an ampersand. And finally, we can combine these into fairly complex queries. In this example, we're specifying the information we do and don't want back. We're asking for the items reviews and images, but we don't want any information about how much is in stock. As you design your API, think about how large the objects you'll return are going to be and if there's any opportunities to limit the amount of data that comes back. Now you may be thinking, okay, I get how to reference different resources by URL, but how do I tell the server what I want them to do? That's where REST verbs come in. Each REST verb maps to a specific action that you're asking the server to perform on your behalf. To highlight this, let's take a look at this table. I have a column that represents a typical UI experience on the web or mobile, its corresponding REST verb, and then what we would expect from the database for that action. Now we'll start with viewing information for which the corresponding REST verb is GET. And we broadly expect the server to perform a database read operation for us. And I want to call out that there's only one GET verb, not a get list or get single. We're specifying list versus single on the object's URL, not the verb. Next, deleting something. The REST verb is also called DELETE, and we expect the database to perform a delete function. Adding content is a little different. The REST verb is POST, which I usually think of as an old timey way of sending a letter by posting it. And this corresponds to a create action in the database. And finally, updating an item. Now this one's a little tricky because there's actually two REST verbs we could use, PUT and PATCH. More on that in a second. We broadly expect these verbs to map to a database update function. Updating and creating data via REST can be nuanced, so let's make sure we understand the differences between POST, PUT, and PATCH. Let's start with the intended use of each verb. POST is straightforward. You use it to create new items. PUT and PATCH are slightly varied. PUT is intended to replace or update an entire object, whereas PATCH is for only partial updates. Think of a few fields on an item rather than the whole thing. Next is idempotency, which is a technical way of saying you can call the same operation over and over again and get the same result. POST requests are not idempotent since they would create a new item each time and therefore would return a different ID of the new item each time. PUT requests are idempotent since you're updating a specific item each time. PATCH requests are intended to be idempotent, but the implementations vary broadly, which brings me to my final point, adoption. POST and PUT are supported by all languages that offer a REST implementation that I've come across. PATCH's adoption is far less consistent. Also, if you look at some of the major public APIs, adoption of PATCH is low and inconsistent there too. For these reasons, we'll be sticking to using PUT in this course for both full and partial item updates. One of the things that comes to mind for me when I'm deciding on how to structure my API are the words of Hal Abelson, a renowned computer scientist at MIT. "Programs must be written for people to read, and only incidentally for machines to execute." I keep this sentiment in mind when I prioritize readability and usability for engineers over sticking to an exact importation of the REST standards, and I'd encourage you to do the same. So we've talked about how to build requests to our API, but what about responding to them? We'll tackle that in the next clip.

### REST Responses and Codes

So we've got a request coming into our API. How do we respond to it? Not only can we send back the data, but there are some standardized response codes recognized by browsers and devices too. Let's start about the actual data that our API will respond with, and there's three main options. First and most common will return the data representing the item or items of the request. This is most common for those GET requests. Next, we might send a status message, something like item created with ID and then its identifier or item deleted successfully, that sort of thing. And finally, we might return just the status code without any content or additional data at all. Now let's take a look at those status codes. Every REST response must have a numeric response code ranging from 100 to 599. Codes are grouped into categories by the hundreds. Now before you start to worry that there are nearly 500 codes to remember, I want to assure you that most of those numbers are unsigned. In fact, only about 75 codes are actually used. And if I'm being honest, of those 75, there's fewer than 20 that are used with any regularity. Now if you want to see that full list, check out the link at the bottom of the screen. So let's start with the 100s. Now these are informational in nature, things like that the server is switching to a new version of HTTP, that sort of stuff. The use of the level 100 status codes is pretty rare. Next up are the 200s, which are used to indicate success, and there's a handful of commonly used 200 level statuses. The first and most common REST code is 200 or OK, and it's used almost exclusively with some kind of response body. And it's really common with GET requests. Next is 201 or Created. And as you might have guessed, this is the typical response used for POST requests. Finally, we have 204 or No Content, and you typically see this with DELETE, PUT or PATCH operations. The 300s are all about redirection. These aren't too common, but there are a couple scenarios you might encounter them. 301 or Moved permanently is a way of forcing clients off of an older version of your API. For example, if you have a breaking change in v2 and you've reached the end of the support window for v1, you could return a 301 status and the URL for the v1 endpoint map to the v2 endpoint as the response body. Some APIs allow for caching parameters to be specified during the request, and that's where 304 or Not Modified comes in. It's essentially a way of saying, hey, you've already got the most up‑to‑date version. Just use the data you have already, and we won't send you anything new. And finally, there's 307 or Temporary Redirect. You might use this response code if you're doing some maintenance on the server and wanted to shift traffic from one to another server. And generally, we don't see this status code used very often. Client errors or 400s are heavily utilized status codes. They're a great way to communicate to clients what we expect from them and how that request failed to meet those expectations. The most common and most generic response is the 400 or Bad Request. I usually pair this with a status message, communicating the parameters that are missing or formatted incorrectly. 401 or Unauthorized is pretty self‑explanatory. You can also provide an additional message like not logged in if you want to be extra helpful, but do know that that comes with a security tradeoff. Forbidden or 403, on the other hand, communicates that the request is properly formatted and authenticated, but the authentication just does not have adequate permissions for the request. Now another incredibly common status code is 404 or Not Found. It's actually so common that it's made its way into popular culture. A couple years back, I made a 404 costume not found for Halloween, and even my non‑technical friends understood the joke. 405 or Method Not Allowed is simply a way of letting the client know that they used an HTTP verb that the server does not support. And this is usually the case if you try to do a modify request like a POST or DELETE on an endpoint that only supports gets. 406, Not Acceptable, and 415, Media Type Unsupported, are two different versions of the same error. Essentially, they're signifying that the client is using a type of media like XML instead of JSON that the server does not support. 406 is where the client is requesting the response at an unacceptable type, whereas 415 is when the client is trying to send request data with an unacceptable type. The final category of status codes are the 500s or server errors. Broadly, these represent minimal information for security purposes. 500 or Internal Server is the most commonly used 500 level status code. For security purposes, I recommend keeping the details of the error to a minimum when responding with this code. Something simple like a correlation ID or how to report this error is usually sufficient. And finally, we have 501 or Not Implemented. This isn't a very common status code, but it's typically used to serve as a coming soon type message. Now in the next clip, we're going to look at the different content types typically used with REST.

### Content Types

REST allows us to specify a wide variety of content types for both requests and responses. In fact, the internet's governing body, the Internet Assigned Numbers Authority, or IANA, has published a standard with thousands of different content types and subtypes. Check out the link at the bottom of the slide if you want to learn more. While the IANA's list of thousands of content subtypes is unwieldy, there are only a few parent content subtypes. I'm not going to go through all of them, but instead I'll focus on the ones that we're most likely to interact with as API developers. Content types are always formatted with their category name, then a backslash, then the type. The first category is Text, indicating that its types are human‑readable. Now if you've done much web development, you're pretty used to these types, specifically CSS, HTML, and JavaScript. Next, there's the Font category, which as you may have guessed is for fonts and typefaces. Again, this is primarily for web development. Now Application is seen as a catchall for binary data that doesn't explicitly fall into any other categories or requires a specific application to use it. We'll be focusing on this category primarily in this course, specifically JSON and XML. And there are a number of media types that are used to share, well, media. The audio image and video types all have subtypes that match well known file formats like PNG and MP4. But how do we use these content types in a meaningful way? In addition to the URL and verb, we can also include headers. Now headers are a set of key value pairs that accompany the request or response with additional instructions. Now let's imagine a client only wants to receive XML data as a response to their request. We simply use the accept header and specify application/xml as the content type. Now I do want to point out that the server isn't obligated to honor that request, as many APIs are pretty strict on the content types they support. Similarly, we'll consider a client that's sending a post with XML data to us. To let the server know what format the data is, we again simply include a Content‑Type header and again specify the value as application/xml. Similarly, when an API returns content in the response body, it should specify what format it's in with the Content‑Type header. Now we've talked quite a bit about clients, but what are some examples? We'll dig into that next.

### REST Clients and Servers

At the beginning of this course, I mentioned that most, if not all, internet‑enabled devices can support REST, and I meant it. Let's dive a little deeper. For starters, what differentiates a client versus a server in REST? Now there's a few things that define a server, and you might be thinking processor power or storage space, even physical size. And while those are important aspects, I like to think of something even simpler. When a client sends a request to a server, it responds. Clients can generally assume that the server will be up and running. Clients, on the other hand, aren't expected to be constantly reachable or even turned on. Typically in REST, servers aren't making a request to the client, but rather simply responding to the request that they get from clients. Now I mentioned clients could be pretty much anything in REST, and your mind probably might've gone directly to computers, maybe laptops, maybe desktops, maybe other servers. And broadly, they are seen as reliable clients. Typically, they handle lots of data and have dependable internet connections. But consider mobile devices like smartphones and tablets. Typically, they're still pretty powerful, but they're much more mobile, and now that often means that their internet connections are less reliable and potentially slower too. This is important to consider when you're designing your API. And if you're anticipating having mobile clients, you may need to build resiliency for your endpoints to handle retrying connections if necessary. And finally, consider the Internet of Things, or IoT. Now manufacturers are putting internet connections in just about anything these days, and wireless chips can easily be the size of your thumbnail and still be really inexpensive. I've worked with people who made internet‑connected grills and other who made internet‑connected monitors for massive excavators. And from blenders to washing machines, there's internet‑connected devices all around you. So what are the main implications for API design? Payload size. You want to keep your responses small since most IoT devices have minimal computing power. Now I won't get too deep into the logistics of servers since this isn't a course on hardware, but I do want to highlight a couple things. As I mentioned earlier, server size or processing power doesn't dictate whether or not a server can support REST. As a matter of fact, those tiny thumbnail sized wireless chips can be bundled with a processor and some storage and be a REST server. I wouldn't recommend running a production build API of it, but you could. The vast majority of REST servers are dynamic in that they are tied to a data source that can change. Now I do want to briefly mention static servers. These types of APIs have a number of implementations from a simple file server that just returns static documents to something more complex like a mock server that uses JSON as a data source for testing, but doesn't let you change that data. I want to wrap up by talking about which language you could use with your REST API. Now broadly, my advice is to just use the language you're familiar with because chances are it supports REST. Python, C#, JavaScript, Go, Java, Kotlin, Ruby and many others all support REST. So so far, we've covered a lot of theory, but how would we go about building an API for a company? We're going to tackle that in the next clip.

### REST Use Cases

Throughout this course, we're going to be working with Carved Rock Fitness, a large outdoor equipment retailer with a sizable technology ecosystem. Now the needs of Carved Rock won't necessarily be exactly the same as those in your organization. However, my hope is that you'll be able to find some similarities between Carved Rock and the challenges that you are facing. Over the years, Carved Rock has grown significantly, primarily through acquiring other companies. And as a result, they have duplicative and sometimes conflicting implementations of the same functionality, for example, multiple different ways of looking up a customer, but different details are returned depending on the implementation. Next, since each acquisition had its own security model, there are a number of different implementations, which both affect the overall security of the platform and make it harder to manage permissions consistently. Similarly, different companies handled API performance differently or didn't think about it at all, resulting in inconsistent and poor performance across the board. Next is supporting the cash register system Carved Rock uses. It's older and relies on XML data. Right now, the API is separated from the other JSON‑based endpoints, and it's frequently out of date when changes need to be made. And the final challenge is that updating the API is extremely difficult. Because there are so many different parts of it, breaking changes often result in downtime or lengthy testing cycles. Now throughout this course, we're going to build an API to address all of these concerns. For the context of this course, Carved Rock has identified three key personas who will interact with this API. First are customers who will access the API through Carved Rock's public e‑commerce site and authenticated mobile app. Next, we have the cashiers who connect to the API through its cash registers. All of the relevant endpoints for them will have to be in XML. And finally, we have the employees in the stock room who will fulfill orders through an employee portal. Fortunately, Carved Rock Fitness has already tackled the challenging work of defining a unified data model for all of their APIs. Now we won't be building out a full retail solution in this course, but rather focusing on three distinct entities that will help us understand how to structure relationships between endpoints. Now I'd encourage you to consider expanding beyond these items if you'd like more practice. First up, we have items, which represent the inventory available for customers to purchase. Next are the customers, which I'm keeping pretty simple so we don't get caught in the implementation details that aren't really related to REST. And finally, we have orders, which are related to both customers and items. Now we're going to dig a little bit deeper into these fields of the data models and their relationships in the next clip.

### Planning out the API

We've got enough context to start planning and designing our API. Now to do this, you can either start the process from the client perspective or the data model end. Personally, I like to start with the data model because I've been a back‑end developer most of my career, but either way is completely fine. So I've got the item and customer data models ready to go. And as you can see, they're pretty basic and only have a few fields each. Orders are a little bit trickier since an order can have many items in it, so we've got the OrderItem table as well. And every order has a single customer, but a customer can have many orders. So the CustomerId field on the Order object will be the foreign key to that Customer object. Similarly, every order item can only have one order, but an order can have many order items, so we store the foreign key relationship on the OrderItem object. And we're following the same pattern for our OrderItem‑to‑Item relationship. Now that we understand our data models, let's start building out the diagram of our API. We'll use it as an implementation blueprint throughout the course. I'll start with Items since they're pretty easy. Let's follow a database CRUD model and map it to REST verbs. We'll need a POST for create, a GET with no parameters to read all the records and another one with an ID parameter to get a single record, a PUT to update the record, and finally, a DELETE to delete it. Now this is a great time to pause the video and repeat the process for customers on your own. Is this what you got too? If not, take a look at the items above. And we'll create the same endpoint and verb configuration for orders too. But what about order items? Rather than using our usual verbs, let's just create POST and DELETE endpoints for them and assume that we'll include the items in the order details endpoint. So we've got our API mapped to our data model. Now it's time to include some personas and see if there's anything we missed. I like to think of this in the context of specific jobs to be done. A customer will get an order confirmation email and needs to see the details of that order when they click on the link. They'll also want to browse the items available on the public site and see a detailed item page. And we also want to give them the ability to see their own profile and edit it too. Now there's some functionality we miss going off of just the data models alone. Customers want to be able to see a list of all of their orders. Now let's create a new subcollection endpoint, under Customers, GET with the id as the raw parameter. Now let's talk about cashiers. It probably doesn't come as much of a shock to you that they'll need to create orders, as well as adding or removing items to those orders. And just like customers, cashiers need to get information about items, and cashiers need to answer customers' questions about their accounts and orders in the store. So they'll need access to the read customers and their orders. Right now, only cashiers can create new customer accounts. Now I mentioned that cashiers need to look up customers, but how do they do that? Well, of course, they can get a list of all the customers, but that could be time‑consuming. So we'll create a separate search endpoint that will speed things up for them. And finally, we have our stockroom employees. Just like customers and cashiers, they need to know about the items that Carved Rock has, but they also need to update and add new items as inventory changes. A significant portion of a stockroom employee's responsibilities is fulfilling orders. And to do so, they'll need to get all the orders and their details. And finally, as an order is fulfilled, they'll need to update its status. Now that was a ton of information, but we've got a great start for writing an API, and we'll jump into writing in that code in the next module.

## Building APIs

### Setting up an API

Let's start building out our API. I want to call out that I'll be using Node.js with TypeScript for this course, primarily because it's a really popular language. Now that's not to say it's the only language to write APIs in. As a matter of fact, whatever language you choose is probably a great fit for you and supports REST too. I've created before and after checkpoints for each module. This will ensure that you always have a good starting point and an easy way to check your work as you go. There are a couple of different ways for you to get this code. Now the option I'd recommend is GitHub's Codespaces, which stores all of your code in a GitHub repository and gives you easy access to switch between the different branches for each module. Now another huge benefit is that you actually get a version of Visual Studio Code in your browser preconfigured and ready to go. All you need is a free GitHub account. I'd highly recommend this option, especially if you already don't have a functioning node environment. I've created this short link for the GitHub repo, and you can find the configuration instructions for Codespaces at the bottom of the README file. The next option is to clone the GitHub repository to your local machine and configure the project there. Be sure to follow all the local setup instructions in the README so the project works. The shortened link is the same for this option as well. And finally, you can download the exercise files for the course page. You can find them in the Exercise Files tab. You will need to follow the local instructions in the README file as well and open up the proper folder for each module checkpoint. I've got the GitHub Codespace for our course pulled up, so let's do a quick tour of the codebase before we jump in. There are two high‑level folders, frontend and server, and we'll be primarily working in the server folder. Within our server folder, most of the changes will take place in the src, or source, folder. The index file at the root of our src folder contains all of the top‑level routing logic of our API. Any endpoint we create will ultimately end up here. Let's run our code using Visual Studio Code's integrated debugger. Click on the Run and Debug button in the JavaScript Debug Terminal. This will open up a new terminal where we'll execute our build commands. I've created a helper command that will switch the terminal to the server folder and then run the server while monitoring for any changes. This will prevent us from having to constantly restart the server every time we change something. I want to call out another important file, our top‑level routes file. I'm storing this inside of our features folder. Node, specifically the Express library used, has this concept of routers to define routing and actions on an API. You can have many routers in your app, and it's a common pattern to have them nested. Here, I'm defining an API router and then telling it to use the item's router anytime someone asks for something on the /items route. You can see me repeating this pattern for the other features in the API, customers and orders. And finally, we're creating a new parent router that routes all of our API requests to the API router above. It also handles get requests to the base URL so that we see a success message when our server loads properly. The last thing we'll do is confirm that our server is actually running. To do that, we'll open up the Thunder Client extension. This is a REST client we can use right inside of Visual Studio Code. We'll start a new request and leave the request verb as GET. We can set the URL to our server URL, localhost:4000, and click send or hit Enter. And we get back a 200 response with a success message. Now in our next clip, we'll start making some of those GET endpoints.

### Getting Data with GET Endpoints

As I mentioned previously, I'm using Node.js and TypeScript for this course. And now while my code examples will all be in that language, that doesn't mean we'll be missing the fundamental concepts of REST. Let's take a look at how C#, JavaScript for Node.js, and Python all define a function that returns a specific order. We see that all three languages define the route that the function will process. Similarly, all three code samples specified that they're looking for GET requests. And finally, all three of them specify that they're looking for an id parameter. Let's start by making a simple get endpoint that will retrieve a list of all of our items. Inside of the items folder, we have a router and a service file. The service file is where all of our database interactions happen. And it exposes simple functions for us to use like getItems or deleteItem. We won't have to make changes to the service file, but you're welcome to check them out if you want. The router file is where we'll be doing all of our work, so let's open that up. Unlike the base router file, this one's pretty empty besides our initial router definition and a helper function that generates image paths that you can ignore for now. Let's start by creating the get list endpoint. We'll assign the /path to our items router and then specify request, or req, and response, or res, as parameters. We also need to make the function asynchronous as our database calls are async. Then things are pretty simple. First, we need to get the items from the service function. And you can press Control or Command period to get the quick fix suggestion for this import. Then we need to loop through our items and populate the image URL for each of them. I'm choosing to do this in the route rather than the service since we don't want the service to know about the API implementation details, then it's just a matter of returning items. Fortunately, Express has a function that automatically converts a JavaScript object to a JSON for us. You can see at the bottom of the screen that the server has reloaded for us, so let's head to Thunder Client to check it out. We'll start with our existing request and add /api/items to the end of the path. And just like that, we get our list of items back. Congrats! You just wrote your first REST endpoint. But what about a specific item's details? Let's do that next. We'll create a new function that responds to /:id. The colon has expressed now that there's a URL parameter to look out for. We'll create a new variable for id. By default, all parameters from URLs are interpreted as strings, so we'll need to parse it as an int. Then we'll call the params property on our request object. Since we've included the :id in the path, we have an id property to reference. If you want to confirm this, rename the parameter in the path or add an additional one to see what happens. With our ID in hand, we can call the service's getItemDetail function. Don't forget to await that function and import the dependency. Now the ItemDetail function is designed to return null if there isn't an item matching the ID, so we need to handle that scenario. So if the item isn't null, we can set its image URL and return it as a JSON object. And if it is null, we can return a 404 with an error message telling our client that we can't find that item. Let's go check out our second API. We still have our items list up, so we can pick an ID from there. Grab whichever one you'd like, and then add a slash to the path, separating it from the items. We see the item's detail response in the description. To check the Not Found path of our code, add a couple 0s to make your ID really big. We see our 404 status working, as well as our error message. Great work! Now it's your turn. Can you create the get list and get details endpoints for customers? Go ahead and try it on your own. Two small hints. You don't need to do anything with image URLs, so your get list function should be even easier. And customer IDs are UUIDs, so you don't have to parse them to ints. Does your code look something like this? And are you getting successful responses from Thunder Client? Let's take a quick look at our API diagram. I've removed the personas for now to keep things more clear. So far, we've completed the highlighted endpoints. And since we're just in the customer endpoint, let's create our order subcollection and search calls as well. Let's tackle the order subcollection first, which is pretty straightforward. We'll define our get request and then start with a /:id just as we did for the details request. Then we simply add /orders to the end of it and finish it off with our req and res parameters. In the function body, we can retrieve the id from the params property of the req as usual and pass through the service's getOrdersForCustomers function. Finally, wrap it in JSON and send it back to the client. Nice and easy. We'll do something very similar for the search endpoint. Create a function call with /search:/query so that we're responding to a search endpoint and expecting the client to provide a query as a URL parameter. And the rest is really easy. Pull out the query, send to the service, and return the result as a JSON. With those changes, we've written all of the get endpoints for items and customers, so now let's focus on orders. The GET with the id one will be pretty easy, but the GET all endpoint will be a little different. There are tons of orders, so we'll want to be able to page through them. There's plenty of scenarios in which we may not want to return an entire dataset like it being too much data to display on the front end or not wanting to overload the server or client. So we'll use this array of orders as an example. Now there's only a handful of items here, but imagine there were thousands. The full list is what we would get from this API call, but let's say that we wanted to get orders in sets of threes. A common pattern is to use take and skip parameters. To get the first page of three, we would include a query parameter named take with a value of 3 and one named skip with a value of 0, which would return the first three rows. Then for our second page, we'd leave take at 3 and set the skip to 3 to skip the first three rows of data. Clients assume that the API is handling scenarios where they ask for more rows than what remain gracefully. And here we can see this happening as we ask for three rows, but there's only one remaining. I've already copied the item's code and updated the imports into our orders router. But our getOrders function needs the take and skip parameters. Let's add those now. First, we need to get the parameters from the request query object. Then we'll need to do a bunch of validation on our query parameters. We'll start by checking if take exists at all, then we'll confirm it the string. And finally, we'll parse it to an int and make sure that it's greater than 0. After all, we don't want to take 0 records on our get call. Let's repeat this process with skip. Switching the parseInt check to be ‑1 since it's possible to skip 0 records on the first page request. Then inside of our if block, we'll pass the parse, skip, and take values into our getOrders function and return the data. And we'll create a helpful message for our client if they fail the validation too. We'll pass a 400 status code and then let them know of our expectations for the parameters. In Thunder Client, we can switch to the orders endpoint and add our query parameters. Go ahead and try a couple different variations, including not passing correct parameters to verify our error checking. Now you may be thinking, that sure seemed like a lot of validation for two simple numeric fields. Are we really going to have to do that for everything that comes into our endpoints? And the answer to that is no. In the next clip, we'll look at different forms of middleware that can be used for validation like this.

### Improving APIs with Middleware

In the last clip, we were manually validating the request data on each endpoint. And to be honest, it was getting a little old. Let's take a look at how middleware, a common API pattern, can help us with this. Imagine that we have our client and server, and the client makes a request. We're going to have all kinds of questions we need to answer before we start doing any of the work on the request itself. For example, does the endpoint even exist or do we need to return a 404? Does our request have the parameters we need? And are those parameters in the proper format? And lastly, is the client even authorized to make this request at all? We'll have some questions we need to address as the server sends out the response too. Are there any errors that were generated? And do we log those errors properly, as well as any kind of universal logging we want to do? Now this is where middleware comes in. It's a code that separates the functionality of specific endpoints and can be reused throughout your API. So we know what kinds of scenarios we can use middleware for, but where exactly can we use it? One of the most helpful and common ways to use the middleware is across the whole API. This is great for error handling and addressing not found endpoints. You can also add it to specific routes like if you want an entire route to require authentication. And finally, you can add it to specific endpoints like our validation scenario from earlier. Let's take a look at how middleware is implemented next. I've actually created a couple of middleware functions in our app already, so let's check them out in our index file. Notice that our call to the routing happens before our call to the middleware. This is important as requests are processed in order. Placing middleware calls before routing causes them to be executed on every call in routing and placing them after it causes them to be executed on any call that isn't properly handled by the routing. With that in mind, let's take a look at our notFoundHandler. The function parameter of all non error middleware functions has the same three parameters, request, response, and what to do next. This is how it's handled in Node. And while the syntax would be different in other languages, you'll probably have similar data and parameters. Also notice that I'm not using the next function parameter. The not found middleware is the last thing we'll register in our API. If a request has hit this function, there's nowhere else for it to go, so there is no next function. And the function body itself is really simple, actually. We're sending a 404 error and a message saying that we can't find the endpoint the client requested. We could, of course, log this failed request, but that's outside of the scope of this course. Let's make a plan on how we're going to implement our validation middleware, starting with the endpoint we need to validate. We'll need a validation tool, which for Node is the popular Zod library. Be sure to check out how to handle validation in your language of choice. In Zod, we'll define what data we expect and what types and finally where it will be located like in a query or a route parameter or in the body itself. Next, we'll pass that definition to our middleware, which we'll compare to the request. If the data matches, the request will be returned back to the endpoint where the validated data can be used. However, if the data doesn't match, we'll return a 400 error outlining what was incorrect about the request so that the client can fix it. Let's start by looking at some Zod validation schemes in our type file, which is at the root of the features folder. At the top of the file, you can see that I've created a couple of definitions already. Specifically, we can validate if the route has a numeric ID parameter or a UUID parameter respectfully. Just below our two ID schemes, I've created one that we'll need for paging to. I've made the decision to preload these for you since the details of Zod aren't critical to this course. You can see that we're asking Zod to enforce the validation requirements we specified earlier. Now that we understand our validation schemes, let's create our validation middleware file in the middleware folder. We can call our middleware functions whatever we want, but I like to stick to something that makes it obvious what it's going to do. So I'm going to go with validate. Next, we need the validation scheme as a parameter. Zod provides us with a parent AnyZodObject type, and we can import that from Zod. Then we can specify the usual middleware parameters, and don't forget to close the function, giving yourself some space to write it. Since all of our calls are in the service are async, let's make this one async too. Zod gives us a safe parse function that allows us to check if the data is correct without throwing an exception. We can also specify all three fields, and it'll just ignore the empty ones. Before we move on, you might notice some errors in req. That's because req is actually a generic interface that we have to manually import the correct one from Express. Not a big deal, but let's actually do it for response too since we're up here. Then it's just a matter of checking if parsing was successful and returning the next function if it was. Now if it wasn't, things are a little bit more complicated. We'll want to return a 400 status error, but we also want to tell our clients what they did wrong. We'll add a details property to our response and then chain the issues of the error properly together so that consumers know which fields has the specific error they encountered. Okay, now let's start using our new validation middleware on the orders endpoint. Because each function will have a different validation scheme, we want to call our validation middleware on each function separately. Between the slash for the route and the async req and res parameters, we can add a new validation function, specifying the paging request scheme. We want to import both of those before moving forward, of course. Now we know that we will get valid data coming to the function, but we still have all of this manual validation code. Gross! Honestly, let's just delete the whole body of the function because we really only need three lines, one to parse the data, notice that I'm not using safeParse since we know that the data is good at this point, another one to take the data and send it to the service to get the orders, and a third one that returns the orders to the client as JSON. If your server isn't running already, now's a good time to start it. First, let's start by calling the orders endpoint in our REST client without any of our required parameters. And we get a couple of errors right away saying that we need both take and skip and that they need to be in the query. Great! Let's add them. And our validation is working properly. Awesome! I'd encourage you to play around with some other validation scenarios like negative numbers, letters, even decimals to make sure we're covered. Let's also handle our ID validation on the other orders endpoints. Just like before, we'll add validate to the function definition, but this time we'll use the idUUIDRequestSchema. Then we can parse our request date again and finally update our call to the REST service. Go ahead and check it out in your REST client to make sure it works. Now it's time to test your skills. Go ahead and perform the same ID validation for the customer's get by ID route. And then when you're done, you can do the same for items. Pay attention to the parameter types for the items endpoint if you get stuck. Now we've got a really great validation framework in place here, which makes it super handy. And our next clip, we'll start creating data with POST requests.

### Creating Data with POST Endpoints

Let's make some POST endpoints that will allow us to create data, starting with items. The first thing we need to look at is the shape of the data we'll ask our clients to provide, which we can find in our types file at the base of our features folder. We do have regular types, but this is what we use to send data to the client from the server. What we need to use instead are specific definitions for receiving data, either on creation or update. These definitions are often called data transfer objects, or DTO, which is why you see that naming convention. The item DTO defined what field we're expecting from the client, a name and a nullable description. Then further down, we're letting Zod know that we need the item DTO definition in the body of the request. Pretty straightforward, so let's write that endpoint. In our itemsRouter, we can start by defining our function using POST, the route of the endpoint, calling the validate middleware and our usual req and res parameters. And just like our GET endpoints, we can use the schema to parse the data. Next, we'll call the item service upsertItem function and pass the body property of our validated data variable. The service returns the newly created item, so let's check if it's null or not. If it's not null, it was created successfully, so we can return a 201 created status code and the item as a JSON representation. But if it is null, something went wrong and will return a 500 server error status and a generic message. Now in real life, we should also be logging this error. Now let's run our code. If you're not running already, start a JavaScript debug terminal and then type npm run dev‑server. Let's start by doing a simple GET request to our items endpoint so we know how many items we have to begin with. Looks like our last item is a headlamp. Now let's change the verb to post and configure the payload. I'll create a bike helmet, but feel free to add whatever type of item you would like. When you're satisfied, hit Send. We get back a response with the name and description we specified, but also an ID. And let's make sure it shows up on our GET call too. And there it is. We've created our first POST request. Go ahead and repeat this code to create new orders and customers too. It's functionally identical to the items code, so it's a great way to practice creating it from scratch rather than just copying and pasting. Let's take another look at our API diagram thus far. You can see we're starting to get things more filled in, which is great, but it looks like we have one more post left to tackle, adding items to an order. Let's go ahead and take care of that now. Remember how we handled GET requests for subcollections? Handling subcollection POST requests is really similar. We'll start by having an ID placeholder in our path just like the GET request, and then we'll validate the payload with our validation middleware. Let's take a look at that validation scheme for a second. You can see that this time we're using two different properties of the object. We're collecting the ID from the param property, and we're also collecting an array of order item DTOs in our body property. This really highlights the power of our validation middleware. We're able to validate data from different parts of the request in a single function call. Now back in the router, we'll build our data properly as usual, and then we'll call our addOrderItems service function. Notice that we can specify the specific ID property in the params, but just include the body property for the items array. That's how we've structured the validation request. We could've made the body object more complicated, but it's best to keep it simple. The service call returns the order with the order items added, so we can do our usual null check and then either return it successfully created or send an error. Overall, adding items to a subcollection is remarkably similar to a regular edition. Once you've written the code, I'd encourage you to head to your REST client and check it out before moving on. Just a quick note before we wrap up. If you're using Git to keep track of your changes and compare your work to the after branch in this module, be aware that the database file is stored in the Git repository and will change as you edit the data. Feel free to ignore the change. Now with that out of the way, what happens when you create some data on accident and need to delete it? We'll tackle that next.

### Deleting Data with DELETE Endpoints

Our API diagram is getting more and more filled out. Let's build out the ability to delete data with our DELETE endpoints, starting with items. The code for DELETE endpoints is pretty similar to a single object GET endpoint. We'll start by calling DELETE on the router and passing in the /:id for the route. Then we'll validate that the ID exists and is in the right format. And once we have our usual request and response parameters, we can validate the data and then call the service. Now this is an interesting decision point in REST. Currently, the service is set up to return the item that was just deleted, which is what we're checking against to make sure it succeeds. Can you think of some scenarios in which it makes sense to return the item to your client that it just asked it to remove? There are a couple other ways you can handle deletes. First, you can return a 200 status code with a message saying something like deleted item with ID, specifying the ID that the client passed in, or you can return status code 204, which means no content. Ultimately, how you handle these scenarios is up to you and the needs of your client. The customer and order delete endpoints are almost identical to the one we just wrote, so take some time to write them on your own. Remember that item IDs are integers, and order and customer IDs are UUIDs. Let's check back in with our API diagram. It's starting to look great. We only have a few more endpoints to write, and our final delete endpoint is on item subcollection of the orders, so let's go write that now. Deleting an item from a subcollection is a little bit more complicated, mainly because we need to have two IDs, the order ID and the ID of the item we want to delete. Fortunately, Node and frankly most languages that support REST are well equipped to handle this. We'll start by a regular delete call and then start building out the path. We'll add a /:id as usual and then follow it with /items. Now here's the cool part. We can just add a second placeholder with /:itemId, and then we do our validation. I've created a special schema that requires a both UUID ID property and an integer itemId property. And then from there, everything is the same as any other delete function other than needing to pass both parameters to the service. And that's it. We've created all the DELETE endpoints for our API. Definitely check them out in your REST client. And if you deleted the wrong items or you just want the data back to where it was, you can just revert the changes to the CarvedRock.db file. Now deleting data is pretty aggressive, but what if we just want to make a couple of changes instead? We'll cover that in the next clip.

### Updating Data with PUT and PATCH Endpoints

We only have our three PUT endpoints left. Remember earlier when we talked about the differences between PUT and PATCH? Let's dig into that a little bit deeper before we get into the code. So here's a quick recap of the differences between PUT and PATCH, and the Carved Rock API will be using exclusively PUT endpoints to update even partial items. While this isn't the literal adaptation of the REST standard, it's a de facto industry standard. As I mentioned before, very few APIs use PATCH endpoints, and there's even a few popular programming languages that don't even have a PATCH implementation. Let's take a quick look at what our PUT endpoints will update on each of our models. On our Item and Customer objects, we'll allow updating of every non‑ID field. In other words, we're updating the entire object, the ideal scenario for a PUT endpoint. Now we only want to update the status on orders, so technically PTACH would be more appropriate, but we'll be explicit that we're only updating that field, and it'll be fine. One more final thing to point out. While you technically can include the ID in the body of your request, it's best practice to include it as a URL parameter as we did for our GET endpoint that returned a single object. Let's go make those code changes. We'll start with the items router. I like to think of PUTs endpoints as hybrids between GETs and POSTs. We need an ID in the parameter, but then we also want to validate the response in the body with the PUT request scheme. Let's check out that PUT request scheme really quickly. We're actually nesting a couple different schemes here, which is what we want for our clients. We have the idNumberRequestSchema, which we previously used in our GET request that requires an ID URL parameter, and we're merging that with our POST request scheme, which contains our itemDTO schema. Back in our itemsRouter, things are pretty similar to POST requests. We validate the data and pass the objects to the service's upsert function. We also need to pass in the ID from the URL as the second parameter. Then we'll go back to the code similar to our GET endpoint, checking to see if the item is null or not and returning it if it's fine and otherwise returning a 404 and an error message. Now we could just not return an item and instead return a 204 status code and an empty body or just a success message. It really depends on the needs of your clients. The customer implementation is almost identical to items, so it takes some time to try on your own. Now let's talk about the orders router. Unlike items and customers, we're only updating a single field, so let's take a look at how this is handled differently. I'm going to go into some nuances here, and I want to call out that these aren't REST‑specific or even node‑specific. They're just how I've chosen to implement the service. That doesn't mean that you have to follow this implementation when you build your own API, and you'll see what I mean in just a second. Things start out exactly the same in our item and customer endpoints, but here's the twist. I need to create a full order object because the service is expecting one. Now again, this is just how I've chosen to implement it. You could just as easily have an update status function that takes the order ID and a new status. Now other than that, the rest of the code is the same. And with that, we've created all the endpoints for the API. Great! Course over, right? Not quite. Remember the requirements for Carved Rock that some endpoints needed to support XML? We'll tackle that in the next clip.

### Supporting Multiple Content Types

So far, we've created all these endpoints, but right now they only support JSON. However, there is a requirement that we need to support XML for the cashiers on their cash registers. That means that we need to modify almost every customer endpoint, the GET endpoints for items and the POST and item operations for orders. Now before we get into how we'll configure our API to support XML, let's briefly look at the major differences between JSON and XML. I've simplified our item list as an example. Starting from the top, we see that XML requires our arrays to be named compared to just using brackets in JSON. This naming is something we get to decide, and it isn't tied to any particular convention. I'm just using items to provide a clear instruction to clients on what they're getting from the API. I could've just as easily called them objects or data instead. Individual objects have to be named in XML too. You can start to see how this is going to get pretty verbose pretty fast. And one final difference is how properties are handled. JSON requires property names to be in quotes and then separated values by colons. XML, on the other hand, doesn't require a name in quotes and uses an equal sign instead. Now this isn't a big factor. It's just something to be aware of. We'll start by retrofitting the GET endpoints for items. We'll need to complete two high‑level tasks. First, we need to check the accept header to see if it specifies application/xml. Otherwise, we can just return the JSON response. Next, we'll build out the items list in XML. Now this process varies heavily, depending on your programming language. C#, for example, you get this functionality out of the box. And note, however, we have to build it from scratch using an XML builder library. We'll start by creating a root element, and we'll call it items, then we'll loop through each entry in the items array and write it to our root item with item as its name. Then we'll return a 200 status code, send the data. We need to close the XML object by calling end, and we can also add pretty print to format it for readability. This adds whitespace, so it would increase the payload size. So you might not want to do this in production. If your server isn't running, go ahead and start it. Let's try out our new code in the REST client. We'll execute a GET request on our items endpoint. That's weird. It's just our JSON payload. Oh, that's right. We still need to specify that accept header. On the Headers tab, we can see that the accept header is currently set to allow anything, so our server defaults to JSON. Let's specify application/xml. Now when we send it, we get our XML payload back. Great! How about an individual item though? We'll start it the same as our list endpoint by checking for the XML accept header. Then our payload is even easier since we don't have to create an array. We can just create an XML item on the fly and send it. And don't forget to put the JSON version in the else block. Now notice that our error handling block is still JSON only. We'll check the accept header again, and then we'll build an error message XML object on the fly. Here, I'm setting the message property on an object called error before sending it with a 404 status code. The JSON error code stays the same. Now let's check this in our REST client. We'll start by checking an item that we know exists like number 3. Notice it's not formatted as nicely. That's what it looks like when you turn pretty print off. It's not terrible for small payloads like this. Next, let's try an item that we know doesn't exist like 30, and we get our error message. Let's head back to our API diagram. We've created our first XML GET endpoints for items. Now the endpoints for customers, including search and the order subcollection, all function exactly the same as the ones we just wrote for items. Go ahead and try them out on your own and then compare them to the after code. We'll tackle the customer's POST endpoint next. The first thing we need to do is register an XML parser in our root index file. This will allow us to parse the XML data client sent us into JavaScript objects that we can work with. I've included an xmlparser library in the project, and we'll configure it with some basic defaults. The response part of this endpoint is pretty straightforward. We need to convert both the success and failure of messages to support XML. Other than the different error messages and status codes, this is essentially identical to our GetById endpoint. With our parser registered, we can rely on the validation middleware for the input, but what about the error response? We also need to make sure that the error that we get from the validation middleware is in XML, so let's fix that up. We'll start as always by checking the accept header. Then we'll build out the root error's object with a message that the validation failed. And then we can loop through the issue's property and the result's error property. For each issue, we'll create an error element and include path and message attributes. Then we'll return it with a 400 error status and build out the root object. Our else condition will just be the JSON version we had previously. Let's head to our REST client to check it out. We'll start by building our POST request to the customer's endpoint. First, we'll set the Accept and Content‑Type headers to application/xml. Then we'll set the body type to XML, and we can add our payload. I'm setting each property as the child object rather than an attribute, as there are some challenges in parsing attributes with this particular library. Once we send the request, we can see the new customers created. I encourage you to try around with the validation by removing one of the properties. The PUT endpoint code is exactly the same as GET after the service call, so you can just copy and paste that on your own. We've got all the code written for our items and customers, and all that's left now are the orders. We're not going to do them together since the POST endpoints are the exact same as the ones we've already done. As for the DELETE orders endpoint, it's the same as the GetById endpoint other than different error messages. I'm confident that you'll be able to get it on your own, and the effort code is always there to help. We've covered a lot during this module, and our API is really starting to take shape. Take some time to think about what else you could add to make the API even more robust. A couple of ideas that I had were to add additional endpoints or even logging. You could also create a middleware that handles all of the conversion of responses to XML for you so that we're not repeating ourselves so much. What other ideas do you have? In the next module, we'll start securing our API. This is an incredibly important topic, so I hope you'll join me.

## Securing APIs

### Understanding Security Options

Keeping your data and infrastructure secure is an incredibly important part of building a successful API. Let's take a look at some common security approaches. One of the most basic places to begin is restricting access to resources or API endpoints. The most common form of this is cross‑origin resource sharing, which allows you to control which domains you'll accept requests from, as well as which REST verbs to support. We'll configure our API for CORS in the next clip. Next, you can use a firewall to restrict access to parts of your API from a range of originating IPs. A common scenario for this is allowing open access to public APIs, but restricting the admin APIs to internal IPs only. Similar, you can configure rate limiting, which restricts how many calls a particular IP address can make to your API in a given timeframe. This is a basic way of preventing denial‑of‑service attacks. Next, we have transport security or protecting information as it travels between the client and the server. At this point, it's pretty hard to find APIs that don't use HTTPS to encrypt their payloads, so it's definitely a must. One small exception is that it's fairly common for you to not use HTTPS for local development. With HTTPS, the payload of the request and response are encrypted, but the URL and any parameters aren't. You should always store sensitive information in the body, not in the URL. And the final area of security we'll talk about its access control, which is probably what came to mind when we first even mentioned security. The first line of access control is authentication or simply can you prove that you say who you say you are? This takes many different forms, including usernames and passwords, API keys, and tokens. We'll be implementing authentication later in this module. After we've determined your identity, we need to now confirm that your identity has permission to the resources you want to access. This process is called authorization, and we'll tackle that towards the end of this module. And I do want to call out that there's plenty more to security than what I've called out here. In the final clip of this module, I'll share some resources to help you keep up to date with this ever‑evolving field.

### Handling Cross-origin Resource Sharing (CORS)

Let's talk about cross‑origin resource sharing, or CORS. In the early days of the internet, cross‑site request forgery issues were really common. Essentially, a victim could log into a legitimate website like their bank or a retailer, and that site would store a cookie with their access token. Then if they opened a malicious website in a different tab, that malicious site could then use their data to authenticate to a legitimate site and impersonate that victim. Now this obviously was not great, so browsers began to restrict all of the content to be from the same origin. Images, fonts, and yes, API calls all had to be from the same domain protocol and port as the host web page. But hosting APIs and other resources on different servers is incredibly common, so CORS was created as a new solution. Here's how it works. First, the browser adds an origin header. This is literally just the domain that's sending the request, so something like https://carvedrock.com. The API is hosted on api.carvedrock.com, and it receives a request and checks to see if the origin is on the valid CORS list. If it's valid, the server responds with the data and sets a special access control allow origin header to true. The browser then receives this header and knows that the response from the API can be trusted. Let's check out how this looks in the code. Be sure to pull down the before code for this module since I've got a few things preconfigured for us. First things first, we'll start a new terminal and run our server. Next, we'll open an additional terminal and start our front end. If we switch to our PORTS tab, we can see two running node services, their ports, and their forwarding addresses. We can preview our front end by clicking the Preview button over the forwarded address, and immediately we get an error. Can you guess why? Let's check out the DevTools and see what we can see. I'm seeing a couple connection refused errors from the home page to our localhost API. And if you don't see any errors, try refreshing the simple browser. If we expand the forwarded address column in the PORTS tab, we can see that our API server's host name isn't localhost, but instead a long set of nonsense. Just a heads up, your domain will be different from mine. They're unique per code space. If you're running the code locally, you likely aren't getting this error. Fortunately, all we have to do is update the base URL in the front end with the APIs forwarded address. Let's copy the address and then edit the environment file in the front end to fix this issue. Here, we'll replace the localhost with our forwarded address. Be sure to leave the /api at the end of the URL. Save the end file and then refresh simple browser. Ah, the joys of development, a different error. But now we clearly see this is a CORS issue, including our lacking access control or allow origin header. Fortunately, there's a CORS middleware for Node and Express that's really easy to use. Given how ubiquitous CORS is, I'd be surprised to hear if there isn't a simple way to handle CORS in your language of choice too. Before we configure the API for CORS, there's one more configuration change we need to make to our code space. GitHub wisely adds its own layer of authentication on top of endpoints it exposes in a code space. While this is great from a security standpoint, it makes testing our changes pretty challenging. Since our data isn't real or sensitive, let's disable the security for our API. Right‑click anywhere on the API row in the port section, select Port Visibility, and change it to Public. Now let's open up our server index file and register that middleware. We want CORS to be handled on every request, so we'll add it before our routes. Don't forget to update your imports too, then finish up by saving the file. What we've done is loaded CORS with a default set of options that essentially allows all traffic to hit our APIs. This makes development easier, but defeats the entire point of CORS in production. I've added this short link to the Express CORS middleware documentation that shows all of the available configuration options. I'd highly encourage you to check it out or look for the equivalent documentation for your language. Now I have noticed some odd caching behavior before, so I'd recommend reloading your code space tab if you're seeing that too. We now see our front end properly pulling data from our API. Also on display, why I'm not a designer or a front‑end developer. All jokes aside, we need to have a serious conversation about securing our user's information, which we'll tackle in the next clip.

### Managing User Identity

One of the most fundamental and frankly easy to do poorly aspects of security is user identity. Let's dive into how to properly handle authentication and authorization. There are a couple key principles to keep in mind when it comes to authentication. First, is the identity of the client making the request. Are they supposed to have access or are they a malicious actor? Next, can you prove that they are who they say they are? And also, can we verify that the proof they're supplying is issued to them rather than them trying to impersonate a legitimate client? And finally, is that proof still valid or has it expired? With these principles in mind, let's take a look at the most common forms of authentication on the web. The first and least secure is basic auth or username and password. This approach is really simple to implement, and even if the passwords are rotated every few months, that's still a lifetime for an API data breach. It's incredibly vulnerable since clients are literally sending their username and password with every request. Some variations of this approach do encrypt the password, but it's still broadly seen as a poor approach. The next option is API keys. Now this is a little bit more secure than basic auth, but still not great. Essentially, you're sending a long set of gibberish with every request. Now the benefit of this approach is that the keys themselves can be incredibly long, thus making them less vulnerable to brute force attacks. And API keys are typically also used in server‑to‑server communication, so domain filtering on firewalls can provide additional protection. But just like regular passwords, API keys are only rotated periodically. Finally, we have OAuth 2, which is widely adopted and considered very secure. One of its strengths is that it shares a large encrypted token that's difficult to crack. And even if you somehow manage to crack that token, they're invalidated after a very short period of time, typically less than 24 hours. I'll go into more detail on OAuth 2 in the next clip. Whatever approach you choose, you'll need to implement it, so let's talk about the three primary ways of doing it. For starters, you just write your own authentication from scratch. Let me emphatically say that this is a terrible idea, and you just shouldn't do it. It's likely not a competitive differentiator for you, and it carries a ton of unnecessary risk. Next, you could look at leveraging an open‑source solution. While this still carries the risk of having to patch and maintain the service on your own, it's significantly lower than writing your own authentication service. And finally, my favorite option, just make someone else do it. No, seriously. Every major cloud provider, as well as many independent companies offer authentication as a service. While there's definitely a cost associated with this approach, the saved effort alone is worth it, not to mention the huge risk reduction. With some basic context, you can at least begin to head in the direction of picking how to handle authentication and how it will be provided. It's important to remember that authentication has massive implications across your technology landscape, and picking a provider might not be up to just you and your use case. You might love how AWS does authentication, but your company uses Microsoft exclusively for everything else. However, in this course, we'll be implementing OAuth 2 using the Auth0 platform. We'll dig deeper into both of these items in the next clip.

### Understanding OAUTH 2.0

Since we'll be using OAuth 2.0 as our security mechanism, let's spend a bit more time understanding how it works. There are multiple different ways of leveraging OAuth, depending on your needs and how secure your client is. We'll be using the authorization code flow since web clients are considered insecure. There are four main parts of this flow, the actions of the user, the web or mobile app, the authentication server, and finally, the API we've been working on. The process begins with the user clicking a login link within the web or mobile app. The web app then requests an authorization code from the authentication server. The authentication server responds with a redirect URL with a login prompt. The user fills out their login information and consents to the web app having access to their information. Notice that the credentials and consent are sent directly to the auth server, and the web app never sees them. The authentication server validates the credentials and sends an authorization code to the web app, and the web app then sends this authorization code back to the auth server and requests an access token. A token with a fixed expiration is sent back to the web app. This type of token is often called a bearer token. And finally, the web app can use the bearer token to access protected resources on our API. Now you may be wondering, are we going to have to implement all of this? And the answer is no, this is a course on REST, not OAuth or front‑end web development. I've created a token generator app that you'll be able to create an account on. And once you've created the account creation process or sign in, the generator will display the bearer token that you can copy. You'll then be able to use your bearer token in the REST client. And in fact, what we'll mainly be focusing on in this course is the API where we'll configure the necessary security for our endpoints. Let's tackle that in the next clip.

### Configuring API Authentication

Now that we know how the OAuth 2 authorization code flow works, let's implement it in our own API. We'll start with our trusty API diagram. I'm leaving the personas on it, as they'll help us to quickly distinguish what should and shouldn't be authenticated. There are a couple of different ways to go through this process, but I'm going to start by going through each of the verbs. For starters, there's almost never a situation where you want your DELETE endpoints to be anonymously accessible, so we'll require authentication on all of them. We generally want PUT endpoints to be secure as well so they can control who updates what. Now POSTs are a little bit more nuanced. Many organizations have some unsecured POST endpoints typically for creating new accounts or guest checkout flows. However, Carved Rock Fitness has decided that only cashiers can create new accounts so it can safely mark all posts as authenticated too. Now this leaves us with GET requests, and there's a lot of variability there. Let's go through them one by one. Both of our GET orders endpoints need to be authenticated since they contain private information about a customer's purchases. And we've got a similar situation for customers. Each GET endpoint has sensitive information that we don't want shared publicly. The two item GET endpoints don't need to be secured since we want customers to be able to browse the site without being authenticated. Notice that every endpoint in the orders is authenticated. We can just move the authentication to the router rather than adding to every individual endpoint. And the same is true for customers. Unfortunately, it's not the case for items, so we'll have to do those at the endpoint level. Here's a quick roadmap of the changes we'll need to make to add authentication to our API. First, we'll create the authentication middleware. This is where we'll do most of our work, processing the token that sent the request and confirming its validity. We'll also need to update the error handler to return a specific authentication error message rather than generic 500 server error ones. And finally, we'll modify our router to include new authentication middleware on the endpoints that need it. Before we start writing the code, I want to highlight one important configuration that I've preloaded for us. In the .env file in the server code, I've loaded the AUTH0\_AUDIENCE and DOMAIN. These two pieces of data are what Auth0 needs to validate the access token. I've committed the .env file so that you can easily access the data, but it's generally bad practice to commit .env files since they often contain secrets or other confidential information. Let's get started by creating a file for our new middleware. I'm calling it auth0middleware since the implementation is tied to Auth0, and we could swap it out later if we wanted to. We'll start by loading data from our .env file and make it accessible to our code. Next, we can start writing our validation method, which is honestly pretty straightforward since most of the work is being done by the express‑oauth2‑jwt‑bearer library. I'm going to call the function validateAccessToken and set it equal to the auth function in the library. We need to pass the parameters issuer‑based URL, which is where AUTH0\_DOMAIN comes in from our .env file and the audience, which is the AUTH0\_AUDIENCE from the .env file. While this library is specific to both Node and Express, there are many libraries like this for most languages. That's all we need in the middleware for authentication, so let's handle error handling next. This is a very simple error handling middleware, so let's add some quick checks on the error. The authentication library we're using will throw a couple specific exceptions if the access token isn't validated, so we can check against those. We'll check if the error is an instance of an invalid token error, and then we'll supply a generic error message and then use the status property of the error as the response status code in our generic error message as the content. We can then repeat this process for the UnauthorizedError exception with a slightly different error message. These changes will provide informative, yet secure messaging to our clients. Let's head over to the routes file and secure the entire orders and customer routes. All we need to do is add a call to validateAccessToken between the path and the router call. And with those two changes, we've secured all of the customer and order endpoints. Let's add authentication to all of the non GET endpoints for items, starting with POST. I'm adding it before the model validation since we want to return an authentication error before a model validation error. We also need to add it to PUT and DELETE. That's the last code change we need to make, so now let's start testing. As I mentioned earlier, I've created a small web app for us to go through the OAuth steps to get the bearer token we need for authentication. Here's a shortened URL to get you there quickly. Click the Sign Up button, and then notice that we're brought to an Auth0 URL. Create an account with the email and password of your choice. We now have a bearer token that we can copy to our clipboard and head back to the code. One quick note, I've intentionally set the duration for these tokens to be really short, 5 minutes to be exact. This will allow us to test the invalid tokens without having to wait 24 hours for them to expire. Make sure your API is running and then head to your REST. We'll start with our GET items endpoints to make sure that unauthenticated access still works. Next, let's take a look at our customers list. Now without any authentication, we should get an error, which we do. Notice that we're getting a 401 status code too. To add authentication, click on the Auth tab, and then select Bearer and paste your token into the Bearer Token box. Once we send the request again, we see that we're successfully getting our customer data back. Now, go ahead and go grab a snack or something to drink while you wait for the 5 minutes for the token to expire, then send that request again. Notice that we're getting a different error message now. Yay. To get a new token, just refresh the authentication tester page and copy the new one. Go ahead and test some of the other endpoints in your API until you're satisfied that it's working. But what about restricting access to certain endpoints to specific groups of people? We'll tackle that in the next clip.

### Setting up API Authorization

Something that's often confusing for folks is the difference between authentication and authorization. Consider a large building like an office space and that there's people that want to get in. Authentication is the process of confirming those people are who they say they are before letting them into the building. For each person that enters, authentication verifies their proof of identity and confirms its validity before allowing them access to the building. Now once we're in the building, this is where authorization comes in. Authorization is the process of identifying which rooms inside the building known people have access to. Because authentication has validated the identity of each person, authorization can safely rely on that proof to provide them access to certain rooms within the building. Let's talk about how we're going to implement authorization on our API. OAuth has this concept of scopes, which allows us to restrict what the user has access to. Now some organizations may also choose to include roles, which are logical groupings of scopes, but they are not directly related to one another. While there are no strict rules on how to format scopes, they're often written as action:object, for example, write:items or read:customers. With that in mind, let's assign scopes to our endpoints, starting with items. We'll use write:items and create:items respectively on our PUT and POST endpoints. Now notice how none of our personas are connected to the DELETE endpoint. That's because Carved Rock isn't planning on implementing this endpoint publicly yet. We'll make it a special scope called a deny, not assigned, and we'll make it very clear to not assign this to any role by accident. We'll follow the same pattern for the other two routes too. Now we could technically create two additional scopes for adding or removing items from an order, but creating an order without being able to manage its items and vice versa isn't particularly useful, so we'll add the create:orders scope to them as well. Now for our read scopes. We don't need to add any scopes for the item GET endpoints since they're unauthenticated, and they'll also have to be unauthorized. We'll use read:customers‑single scope for both the GetById and GetOrdersByCustomerId endpoints. And we'll use read:customers for both the GET list and search customer endpoints. Finally, we'll follow the same pattern of read:orders for the list endpoint and read:orders‑single for the GetOrderByEndpointId respectively. With the scopes defined, let's take a look at roles. Just a reminder, roles are a convenience function inside of your authorization management tool. They are not a part of the official OAuth authorization flow. First up is the Stock Room role, which assigns orders and item scopes that we need. Next, we have the Customer role, which assigns us the scopes we need to read a single customer, perform customer updates, and read a single order. And finally, we have the Cash Register role, which has all of the available customer scopes, as well as the ability to create orders and look up single ones. Here's a roadmap of the changes we'll need to make to add authorization to our API. First, we'll have to configure all the roles and scopes in Auth0. I've already completed this for you. Next, well need enums of the available scopes, and I've completed this in the config folder of the API already as well. Then we'll create the authorization middleware. We'll take the scopes as parameters and compare it against the scopes in our token. We'll also need to update our error handler to return a specific authorization error message. And finally, we'll modify our routers to include new authorization middleware on the endpoints that need it. Let's get started by updating our Auth0 middleware. We'll create a new function called checkRequiredScope, and we'll take the required scope as a string parameter. Then we'll forward that to the requiredScopes function in the express‑oath library, which will handle all of the scope checking for us. You're probably thinking, but we didn't really write that much authentication code. And you're right, we didn't. And that's very intentional. Security is one of those things where it's almost always better to rely on the best practices rather than trying to write your own version. Now, let's add some error handling for improper scopes. We're going to follow the exact same pattern as the authentication errors, but with the InsufficientScopeError exception type instead. Go ahead and give it a try before I paste the code in. Your code will likely look similar to this, but maybe with different texts for the error message, and that's totally fine. Now earlier, I mentioned that I previously defined the scopes in the API as enums. You can see them all listed here in the permissions file. The format is very straightforward. Each enum is named object permission, and then each enum entry is with a specific action like Write, Read or Create. Now let's start adding the scope checks to our endpoints. Remember that since each endpoint requires a different scope, we can't do this at the router level like we did authentication. I'm displaying the mapping of the scope to endpoints on the top right of the screen. We don't need any scopes for either of the GET endpoints since they're unauthenticated, so we'll head to POST. We'll add a call to our checkRequiredScope function after validateAccessToken, but before we validate the schema and pass the ItemsPermissions.Create value. And that's it. For DELETE endpoint, we'll use the Deny value of the SecurityPermissions enum. And finally, for PUT, we'll use the Write permission. We've completed all the scope checks for items. Go ahead and pause the video so you have the mappings for the orders and customer endpoints, and do those on your own. Once you're done, resume the video, and we'll talk about testing our work. I've logged out of the authentication web app so we can test authorization. In the real world, employees would have to sign up for accounts or some kind of process would grant them access to the Stock Room or Cash Register roles. However, I don't want you to have to wait on me to assign you a particular role so you can test your work, so I came up with a clever workaround. As you're creating a new account, append the plus sign and then stock or cash for the respective role that you're interested in testing with your email address. These separate accounts can use the same email address if you'd like. When the account is created, I've got some code running on Auth0 that will automatically assign you to the correct role. I'll log in with the cash register account, and we'll copy the bearer token. Before we start using this token, I want to show you a helpful tool, jwt.io. In here, we can paste our token in, and it will decode it for us. I do want to call out that this is strictly processed on the client side and isn't shared with any external servers. In the decoded payload, we can see a number of important details, starting with expiration. Further down, we see the scopes that are part of this token. Here, we can see all the scopes that we would expect with the Cash Register role. This page is great for confirming your bearer token contains the information it needs. Back in our code space, let's make sure the token gives us access to what we're expecting. I'll be using my cash register token, but if you created a stock room token, your results will obviously be different. Once you've started your server, let's send a GET request to our customers list without adding our bearer token. As expected, we get an error. Now let's add that bearer token, and we see our customer data coming back. Awesome! The Cash Register role doesn't give us permission to list all of the orders, so we'll confirm that our insufficient scope exception works. In here we see our Permission denied message that we added to the insufficient scope exception. Finally, let's check out an endpoint that doesn't require authentication and make sure that the bearer token doesn't cause any issues. And as we can see, we get all of our items from our unauthenticated endpoint just fine. Authentication and authorization are big parts of security, but not all of it. We'll wrap up this module by talking about some additional areas to look for in the next clip.

### Improving Security with OWASP

We've covered some of the biggest aspects of securing an API, including resource access, transportation security, and access control. For better or worse, the security landscape is ever changing, and it can feel impossible to stay ahead of malicious actors. However, there is a global nonprofit whose mission it is to provide developers with the necessary tools to create secure online experiences. The Open Worldwide Application Security Project, or more commonly known by its abbreviation, OWASP, is an incredible resource. Let's jump right into the OWASP site, which at first seems a bit overwhelming. There's two main resources I want to highlight. First is the OWASP Top 10, which, as you may have guessed, is the top 10 of the most impactful security vulnerabilities across the web. As I'm scrolling down the page, we see broken access control at the top of the list. Good thing we worked on that in our API. Each of these categories include specific examples of a vulnerability, as well as how to fix it. Now that's great for the entire web, but there's a resource specifically for REST APIs. If we search for API, we get the OWASP API Security Top 10 2023 project. Now this is an amazing resource specifically for API security vulnerabilities. And here again, we see a number of issues related to authorization. OWASP is an amazing resource, and I encourage you to explore it on your own. And if you find the content overwhelming, there are some great courses here at Pluralsight to help you make sense of it. Now in the next module, we'll talk about API performance. I hope you'll join me.

## Improving API Performance

### Identifying API Bottlenecks

Have you ever been on a website that was just so slow? How about a mobile app that just spins and spins until it eventually loads? A poorly performing API might be part of the problem. In this module, we'll take a look at some common causes and solutions to bad API performance. A great place to start is to make sure your API design matches your client's needs. Here's a quick story. I was working with an API to get the last three years of lab results for patients in their medical portal. Unfortunately, the API didn't allow me to specify a date range, but instead only provided paging by the number of results returned. So, we had to fire off 50 to 100 concurrent requests to the endpoint with different paging parameters and then aggregate all of them until we got the full date range. And we had to do this for every visit to the lab results page. If the patient didn't have too many labs, we'd end up with 7 and 10 years worth of data and needed to throw most of that away, or if they had a lot of labs, we'd have to make more requests to get their full 3 years of data. And this was our most popular feature on the site, and it was taking several seconds for the page to load, not to mention all of the stress it put on the API we were calling. So what did we do? We sat down with the team who owned the lab data API and explained our use case. They were able to design an endpoint that allowed us to get historical data without having to chain all those calls together. Now your API might contain some tasks that take a long time to run too. Let's take a look at a couple scenarios and how to address them. Maybe you've got a report that takes a couple minutes to build. You could run that task on a schedule, and then you could cache those results. And then when you get a request for the report, you can return the cached copy immediately, but be sure to include a timestamp so your client knows how fresh that data is. What about a large amount of data that takes a few hours to execute? There are a couple ways to handle this change. When you respond to the request, the data will include information about the task like when it was started and what all was included and maybe even information on how to cancel that request. Now your response can provide a URL for the clients to monitor that reflects the progress of this task. The disadvantage here is that your client has to periodically check that URL. Alternatively, you can ask the client to provide a URL in the request where your API will send its progress to. This is commonly referred to as a webhook and really only works if your client is a server since we expect them to be online and available whenever we're ready to send them the status of the task. Something else to be aware of is making sure that the hardware you're running your API on matches the needs of your clients. I was working on an API that had just launched, and we were encountering a ton of performance problems. API calls that were supposed to complete in under a second were taking closer to 10 seconds, and we just couldn't figure out why. After some digging, we took a look at our cloud server configurations. It turned out that we were trying to serve thousands of concurrent requests on servers that had less computing power than most of our laptops. With a couple quick changes, we were back up and running, and the API was performing as expected. Now I do want to call out that this is a delicate balance. Server resources are expensive, and you need to intentionally balance demand with financial constraints. Another common issue is returning large amounts of data. Depending on your client's network connection and the device's processing power and available storage, massive payloads can be a problem. This is another opportunity to connect with your clients and understand their needs. Are you sending too much data? Do some clients only need a subset of the response? And if so, this is a great opportunity to reduce the size with query parameters. Depending on the scenario, compression can be helpful in reducing your payload size in transit, but be aware that both your server and client will need processing power to compress and decompress that data. We'll dig into compression more deeply in the next clip.

### Using Compression

Compression is a great way to improve the performance of your API by identifying statistical redundancies in your data and removing them in a way that allows the original data to be reconstructed later. In other words, it makes your data smaller. Now there are a number of places compression can take place in your API. The most effective and yet least obvious place is the network infrastructure like Cloudflare or a content delivery network. These servers are the closest to your users. Next are the servers your API runs on. It's very straightforward and commonplace to convey your compression at the server level so that all traffic leaving that server is compressed. And finally, you can add compression to your code. I actually debated demoing it for this course, but ultimately decided against it since it's not very common and it's less effective than doing it on the server. Operating under the assumption that I've convinced you to use compression, the next thing you need to know is which library to use. Now gzip has been the de facto standard in file compression since it was released in 1992. Brotli, however, has gained significant ground since its release. Also a fun fact, Brotli is actually German for bread roll. When gzip was built, the internet was still in its infancy, so it's primarily file system‑based compression, whereas Brotli was developed explicitly for communication on the web. Gzip enjoys broad adoption, as does Brotli, which is implemented on all major browsers, as well as the major operating systems on servers. Brotli's compression is significantly better than gzip's, anywhere from 15 to 20%, depending on the file type. Now unless you're dealing with a hyper‑specific use case that requires gzip, I recommend using Brotli. But what happens if we want to reduce the number of API calls we have to respond to altogether? We'll look at that next.

### Configuring Code-based Caching

If you're not familiar with caching, here's a quick overview. Let's say we have a mobile app that's interacting with our server. The app makes a GET request to the items endpoint, and the server makes a request to the database, and then the database returns the data, and then the server sends the response back to the mobile app. Pretty simple, right? But what if our database calls are more involved or really expensive to make and don't change that often? Let's see how caching can help. We'll make another request for items in the app. The server, again, sends the request to the database, the database then takes that response and stores it in a cache, and the cache sends the data to the server, which sends it back to the client. Let's call the items endpoint one last time. The server checks the cache, finds the data we're looking for. We call this a cache hit, and then the cache returns the data to the server, and the server returns it to the mobile app without having to involve the database at all. Let's go to implement this for our actual items endpoint. If this is your first time looking at the code, check out the building API's module to get set up. We'll be working in the item service since we've abstracted all of the data management functionality of the services file. I've already included a caching library in the project called TTL cache. TTL, or time to live, is a common caching pattern, which controls cache expiration by defining how long before a cache entry is considered stale. First, we need to define how long we want an item to live in the cache. And since our items don't change very often, we'll say a day, which is a pain to express in milliseconds, but here it is. Next, we'll create our cache object, as well as a key for our entry for all items. Let's implement the cache for our getItems function. First, we'll get the entry from our listKey. If it's defined, we can just return it. Otherwise, we'll continue with our database request below. We also need to chain populating the cache to our database request. And this is pretty straightforward. We'll just use the cache.set function and then return the items after that. Now the best way to test this is to add breakpoints to each of the returns in this function and then run the debugger, then we'll call the API endpoint with our REST client. We can see on our first run that we're hitting the database call breakpoint, which is what we'd expect. And we'll continue to the code and run it again, and now we see that we're hitting the cache breakpoint. Excellent! Go ahead and repeat this process for the getItems details function on your own. One hint. Since we'll be storing lots of different items, it's best to use the item's ID as the key in the cache. Does your ItemDetail function look something like this? Now we successfully loaded our cache, but what happens when an item is updated? Let's handle that next. In our upsertItem's function, we need to chain the cache updating to our database call. Fortunately, this looks almost identical to the getItemDetail's call. And we'll also delete the all item's entry since it would have an incorrect item in it at that point. And then we'll finish it up by returning the item. Deleting is pretty straightforward too. Notice that I'm gracefully handling situations where we can't delete an item that doesn't exist by returning null. We'll need to account for that in our caching logic too. We can chain off the database call again and then check to see if the item we received from the database is null. If it isn't, we can delete that item from the cache, as well as the list. Then regardless of whether or not the item was null, we'll return it. I'd encourage you to run through a couple test runs to confirm all of the endpoints of the items router are working. You'll have to disable authorization on the delete endpoints to get that to work, however. So we've gone over how caching works in code. But to be honest, most people just use external caching providers. We'll talk about that next.

### Leveraging Caching Providers

You may have heard of Redis or similar caching providers, but how do they actually work? Now this process is actually fairly similar to our example from the last clip. Essentially, rather than the cache being something that's hosted on your servers, it's a dedicated server that specializes in a highly performant cache. The other way caching providers can help is to offload traffic from your servers entirely using a content delivery network, or CDN. In this model, you cache the entire response on the CDN, and clients never reach your servers. There's a lot of configuration options available, and most cloud providers have CDNs or caching options available out of the box to help you manage performance. Think about your API. What are some areas that change rarely that would be a good fit for either of these approaches? Are there any endpoints that remain relatively static? But what about making sure your API can handle requests that can't be improved with caching? Let's talk about that in the next clip.

### Scaling API Infrastructure

Remember the story I shared earlier where we handled performance challenges by increasing the capacity of the server? That's one way of handling those challenges, especially when the server you're using is legitimately underpowered for the typical workload. This is commonly known as vertical scaling. There is another option, however, especially if your server can handle the load under normal circumstances. Horizontal scaling is simply adding more servers of the exact same size, configuration, and deployed code to your environment. In order to make this happen, you'll also need to implement a load balancer, which will distribute traffic equally across your servers. Cloud providers like AWS and Azure offer autoscaling to allow you to scale horizontally on demand. As a matter of fact, they can be configured to automatically add or remove servers based on the load your API is facing. It's a pretty awesome feature. One quick caveat on horizontal scaling and balancing loads. In order for your API to take advantage of this functionality, it must be stateless. In other words, your code needs to be written in a way so that every request from a client can safely be executed on any server in your cluster without any context from previous requests. Fortunately, that's a fairly common pattern in modern API development. So, we've covered how to support the growing demand of our API, but what about new features? We'll cover that in the next module.

## Managing API Changes with Versioning

### Understanding Changes and Versioning

Any successful organization will grow and evolve over time, and the APIs they produce are no different. Let's take a look at how to manage these changes. When we're talking about API changes, it's all about the impact to your clients. With that in mind, there are two high‑level groupings, breaking and non‑breaking changes. W'e'll start with the simpler of the two, non breaking. Broadly, these are changes that do not require your clients to make any changes unless they want to consume new functionality. One example might be adding additional data to a response of a GET request. It's also common to add brand‑new endpoints as non‑breaking changes. Depending on how significant the added functionality is, you may choose to handle the change as breaking in order to better communicate it to your clients. You may have guessed, but the other type of changes are breaking changes. These types of changes represent a loss or significant change in functionality or any time a client needs to make changes on how they consume your API. In addition to endpoints being removed, another common example is clients needing to supply different data to their POST or PUT endpoints. And the final example is where the structure of a response in a GET request changes. This could be changing the name of a field, its type or nesting data in an array or object. With that in mind, let's take a look at some of the changes we have coming up for the Carved Rock API. On the item model, we need to add a new field for the staff review of the item. Since only some of our clients will need this data and they don't need to change how they call the GET endpoint, this is a non‑breaking change. There's also been a request for smaller images on the list endpoint. and we currently only have a single URL field. So let's split this field into two different ones, thumbnail and full. Once this change goes into effect, clients will no longer have access to the imageUrl field and will instead need to use the two new fields. Because of this, we'll consider this a breaking change.

### Building a Versioning Strategy

Let's talk about different strategies available to us for versioning and how to message that to our client. For each approach, keep an eye out on the gauge to the left of how prevalent it is in the industry. The first and most common approach is to simply include the version in the URL. The typical pattern is to have it be relatively early in the path such as the example below where it's between API and the endpoint items. This approach can have implications, particularly how you organize your code in the API, however. Next, we have a fairly uncommon approach, which is to pass the version as a query string on the request. This requires query validation on every endpoint. I've rarely seen this approach out in the wild. Next is a custom header usually starting with X‑. I would say this is the second most common versioning approach, and this is pretty common to handle it with custom middleware. And finally, creating a custom accept header is something that some people do too. Here, we can specify application/vnd for vendor and then the name of your custom implementation. I'm using carvedrock and then the version followed by json or xml. I'll be honest, I hesitate to even include this approach at all. It's so rare. You might encounter this on a pretty old legacy API that you want to consume. But I wanted you to be aware that it exists, but it's really not a very common option for new APIs. To match current industry trends, we'll be using the URL versioning in this course to support breaking changes.

### Implementing Non-breaking Changes

Before we begin, be sure to check out the before files for this module specifically. I've made some changes in the code that we can use to make versioning work more easily. Now historically, our code has looked like this, a series of feature folders like customers and items and then the types and the route files at the root of the feature folder. We'll be switching the structure around slightly. I have a folder for each version, starting with v1. And then inside that folder, we'll have all our feature folders just as before. We'll also move types and routes for that version into the v1 folder. But we'll also retain a route file at the root of the features folder, and this will handle all of the top‑level routing to v1 and feature versions. So with the context of our new file structure, let's add the staffReview field to the item details endpoint. First, a bit of artificiality. I'm using Prisma, a popular object‑relation mapper for Node to handle all of our database interactions. Here, you can see the model definition for items, and notice that I've already added a staffReview field to our database, and I've also populated that field in the table with some data. Because of this, we won't need to make any data changes. After all, this is a course on REST, not databases. I'm also suppressing the staffReview field in the service for this demo since Prisma automatically pulls all of the data for the table row, and JavaScript would add the staffReview field to our object, even though it doesn't exist in the type definition. So let's fire up the server and see what the ItemDetail's endpoint looks like now. For starters, notice that I'm doing a GET request against the v1 version of the endpoint. There isn't a v2 version yet. And if you make the request without v1, you'll get a 404. Now if we check out the ItemDetail's endpoint, we see the current state, an item with no staffReview field. Fortunately, the way we've configured this API makes it incredibly simple to make this change. A router is simply returning an ItemDetail model from the service to the client and doesn't really care about how that model is defined. So all we need to do is add our new staffReview field to the model. And since the data is already stored in the database, it should just pull all the way through. And we'll delete that select line from the service too. Once the server has restarted, we'll head back to our REST client and send the GET request again. And just like that, we have the new data. Now if you'd like to confirm that this change, in fact, doesn't break anything, I'd encourage you to fire up the front end and make sure that everything loads properly. Check out the Handling Cross‑origin Resource Sharing clip in the Securing APIs module for set‑up instructions for the front end if you haven't set it up previously. And don't forget to change the API forwarded port to public and add v1 to the URL in the front‑end environment file. So this was a pretty simple change. Let's handle something more complex and a breaking change in the next clip.

### Implementing Breaking Changes

Let's talk about breaking changes, specifically removing existing fields and replacing them with new ones. First, we'll create the folder structure for the new version of the API. Then, we'll create our new models or types to match the new fields we'll be adding. I'm also going to create a new copy of the service for our items. Now, this isn't strictly necessary since we'll be making all of our changes in the router, but it's best to have a single set of routers and services contained within a version. And finally, we'll make some real code changes in our new items router. We'll wrap things up by mapping a newly created v2 routes file and then mapping that to the root level routes file. So this is how our folder structure currently looks. We're only making changes to the item's endpoint, so let's make sure to represent that in our tree. And we'll have our routes file point to the new items router. We could make copies of the router and service for each of them in v2 or just have the v2 routes point to the v1 version. Now this isn't ideal long term, as it decreases code readability, but it works in a pinch. Let's get started. First, we'll create our v2 folder in features, and then we'll add a new types file in it. Now all we're doing is changing the definition of the Item and ItemDetail models, so let's copy them from the v1 type definition. Once they're pasted in, we'll change the imageUrl to a thumbnailImageUrl on the Item property. And then we'll add the fullImageUrl property to the ItemDetail model. Next, we'll copy the items folder from v1 to v2. Let's quickly configure our service. First, I'll rename it to items.serviceV2 so it's easier to keep track of. And then, we have the file itself, which we see that is our v2 file doesn't have the ItemDTO, which we knew since we didn't reimplement it, so we need to update all of our imports. The easiest way I've found to do this is to just delete the bad import, find the first instance of the error, and then use the VS Code suggestion to fix it. Now let's get to work on our router. We'll rename that file to V2 as well. Once we open our new file or create it with some errors in our imports again, we can fix these by simply pointing the imports to the v1 types. At the bottom of our routers file, we have the code that generates the path to each item's image. If we expand our public folder, which hosts our static assets, we can see that we have the full size images at the root of the image folder and the thumbnails nested inside of it. Now that we know the structure of the images, let's modify the buildImageUrl function to dynamically route to the thumbnails folder. First, we'll add a thumbnail parameter, and then we'll conditionally add /thumbnails to the path if our parameter is true. With that in place, we can update our cost of this function throughout the router, and our GET function will need to update the thumbnailImageUrl property, so let's do that. And further down, we'll need to update the ItemDetails call with that same code, so the thumbnail URL is populated as well. And we also need to add the full image property with the thumbnail value set to false. Next, we'll build out our v2 routes file. It's easiest to copy this from v1 and then make changes. We've got issues on our imports, as expected, so let's go through them one by one. We changed our itemsRouter name to v2, so let's update that. And we need to update the customer and order imports to point at v1. We also need to rename the v1 router to v2 so it's clear when we're exporting. And the final thing we need to do is register this v2 routes file with the one at the root of the features folder. This is a pretty simple change. We just copy the v1 reference on line 7 and bump the numbers. Don't forget to import it. Now with that, we've completed our breaking change, so let's run it and see it in action. We'll first use our REST client to check the v1 items endpoint. As expected, we see the imageUrl pointing to the full size images for each item. That's a surefire way to have a slow product page, so I'm glad we're fixing it. Now, let's bump that URL version to 2. You can see that we're sending the thumbnail image back, and it has the thumbnail in the URL to point to the client with the smaller images. I'd encourage you to run the front end and see what happens if you try to consume the v2 API without updating the front‑end code to match. So far, we've talked almost exclusively about local development, but what can the cloud offer APIs? We'll talk about that in the next module.

## Improving APIs with Cloud Providers

### Understanding Server Offerings

So we talked quite a bit about building an API, but what about where it's hosted? While massive onsite data centers are still popular in some industries, many organizations are moving to the cloud. Let's dive a little deeper into the capabilities. Now this is a course about REST, not different cloud providers, so I'll only be covering them at a high level. While the cloud marketplace has many competitors, I'll be covering the features that are common across the big three providers. Amazon Web Services, commonly known as AWS, remains the largest cloud provider in the world. And at the end of the first quarter of 2024, it controlled just over 30% of the global cloud market. Microsoft Azure platform continues to close in on AWS, representing a quarter of the world's cloud computing power. And rounding out the top three providers is Google Cloud Platform, or GCP, with just over 10% of the market share. If you did some quick math, you noticed that while these three providers are massive, they aren't the entire cloud market. There are many others. I'm just choosing to focus on the most common ones. Now every provider will have a slightly different name for their products. Azure and GCP typically follow a descriptive naming convention like Azure SQL or Compute Engine. AWS chose to go a different route, naming their products after seemingly unrelated and non‑technical words like Beanstalk or Snowcone. If you're looking for a fun icebreaker with some technical peers, just start saying random nouns and then have people guess what the associated AWS service is. Now because of these distinctions, I'll be talking about the different pieces of infrastructure by their functional name like database or load balancer rather than their provider‑specific name. Let's talk about what goes into deploying and hosting an API. We've already talked about the basics like a content delivery network, or CDN, load balancers, and servers, but what else do we need? We've previously talked about a database, and we'll likely need some file storage for images and other static assets. We need something to manage authentication and authorization for us, as well as caching for performance purposes, and it's really important for us to have robust logging and alerting infrastructure to help us diagnose and address issues too. And we'll need a place to store secrets like API keys. And don't forget, we need a way to build and deploy code too. And finally, we have traffic managing and networking between all of these entities. Now of course, this is just illustrative. Your API may not need all of these items or it might need all of them and more. But what about these items can be moved to a cloud provider? Literally all of them. Each of the big three cloud providers have a specific implementation for each of these components. Now if you find yourself in an environment with a heavy on‑premise presence along with the staff to support it, you'll likely encounter resistance to moving to the cloud. I once worked in such a place and was able to make progress with the infrastructure leadership by framing this as an opportunity for us as an organization to focus on what differentiates us from our competitors rather than dedicating resources to commodity items. It also helps to highlight that Google, Microsoft, and Amazon all have significantly larger budgets and staff to solve the same problems we were trying to solve. So here's a high‑level balance of the pros and cons of the cloud. And keep in mind that you don't have to move everything into the cloud. There's plenty of organizations that only move some of their infrastructure there and keep some on‑prem. One of the biggest benefits is just how easy it is to implement something. Provisioning a server might take you weeks on‑prem, but a matter of minutes in the cloud. When I worked with on‑prem infrastructure teams, I'd have to submit a request with my hardware specs. They would send me a quote back, I'd have to approve the request, then they'd order the hardware, install it, and then a month or two later, I'd have the server. The same process takes about 5 minutes on a cloud provider. Wow, cool server infrastructure, said no non‑technical person ever. Leveraging the cloud allows you to focus on what differentiates your organization from the competition. In the example I mentioned earlier, we did eventually move to the cloud as an organization, and the folks that were used to manage our servers, we found opportunities for them to configure the software that was critical to our organization's success on the cloud infrastructure. And finally, as I mentioned, it allows you to rely on the significantly larger financial and human resources of your chosen cloud vendors. All of the large cloud vendors are in constant competition with each other and are looking to add more options and make them faster and cheaper. Now on the other hand, the biggest concern of the cloud is cost. It's incredibly easy to run up a massive bill, especially if your servers aren't sized appropriately. And if you configure your servers to autoscale output demand, it's really easy to have your bill balloon way beyond what you're planning on paying. Fortunately, many vendors provide a number of cost management features such as alerting or caps. Now each of these cloud vendors also have their own implementations, and migrating from one to another is pretty difficult. AWS actually went as far as to invent its own language to define environment variations. There's some tooling to help you with it, but migrating from one provider to another is pretty limited. Switching providers is often a painful and manual process. And finally, no matter how hard you try, you might still encounter people that believe data is more secure on servers they can physically touch rather than the cloud. While AWS, Azure, and GCP are all likely significantly larger targets for malicious actors compared to your organization, so are their security budgets. While the size of security teams within these organizations is tightly guarded, it's certainly significant. Now earlier, I mentioned cost as a concern, and a lot of that can come from resources sitting idle or being underutilized. In the next clip, we'll take a look at serverless functionality as one of the ways of addressing this concern.

### Shifting to Serverless Functionality

Let's talk about a relatively new option in the cloud, serverless functionality. Broadly, there are two categories, containers and functions. Containers essentially let you spin up a virtual machine without managing any of the underlying infrastructure. If you've done any DevOps work before, you probably associate containers with Kubernetes or Docker, and those are absolutely examples of serverless containers. However, the other implementations are purpose‑built apps like a web app, an API app or even static code. The token generator we used earlier in the authentication module is using Azure's serverless web app environment. The really cool serverless option is functions. In this environment, you literally just write a function in the language of your choice and then define what events will trigger it in your cloud provider. Then, when that event happens, your provider executes the code. You literally don't have to configure anything beyond that. Functions can interact with other cloud services like pulling data from a database or writing to storage. Let's take another look at our infrastructure and see how to leverage serverless functions. We just install the serverless, and then we're done. Okay, not quite. While serverless functions still interact with all of our underlying infrastructure like databases and secrets, the code is structured quite differently. For functionality handled by serverless, we don't even need a load balancer anymore. Serverless functions are also able to interact with other cloud features very easily. They typically have SDKs that allow them to connect directly, making the code simpler to write. And finally, since much of the communication between the cloud components is simplified, your function acts as the front door for your API, and so you can reduce some of your networking configuration. Let's take a look at some of the major differences between server‑based applications and serverless functions. I do want to call out that I'm not going to do a server‑to‑serverless container comparison since how you write the code for them is pretty similar. As you saw in the previous module, servers respond to all traffic on a particular domain or root URL based on a single entry point. With serverless, however, you need to configure a function for each endpoint and potentially each verb, depending on how you design your code. This can make for some pretty complex configurations or really long switch blocks. Now it's quite easy to share code between the different endpoints on a server. Literally just point them to the file's location and import the functionality you're looking for. Serverless functions have come quite a way in this space, and now it's fairly similar to share code between functions, depending on the cloud provider, but it's definitely not as clean and obvious. A big benefit of servers over serverless functions is that they're always on. Essentially when a client calls your API, it begins responding to their request immediately. Since serverless functions are only provisioned when they're receiving requests, they may have a cold start period. Let's break that down for a second. Essentially, once your serverless function is done responding to a request, it begins to decommission the resources it was running since they're not actively being used. Once this process finishes, your function is now cold. When the new request is made, the cloud must reprovision all of that infrastructure for a code to run. Now this cold start time represents the duration of the reprovisioning. Cloud providers are diligently working on shortening this duration with current times averaging around a few seconds or less. So, why go serverless, especially when it's a pretty big shift and can seem harder than server‑based development? Cost. You're paying for a server 24/7, regardless of whether or not your clients are making requests. Serverless functions, on the other hand, are charged only when they're running. If there are no client requests, your functions don't incur any charges. This can represent significant cost savings if done correctly. In the next clip, we'll take a look at one of the most underrated cloud functionalities, API management.

### Using API Management Functionality

We have this high‑level overview of how APIs function, but what about the broader management responsibilities that come with it? The reality is that writing API code is just part of running a successful API. Let's talk about the management aspects, keeping in mind that we've already covered some of them. As your API becomes more complex, you may need to separate functionality across multiple servers or clusters of servers. Keeping this routing consistent and well managed can be a challenge. We've already talked about authentication and authorization on an API level, but what about across multiple servers? And we briefly spoke about rate limiting, but how do you actually implement that? Now there's many approaches, including by IP address, by user, and other access control measures. Caching can also easily grow in complexity and needs to be managed carefully. Now, while there are some logs that should happen within the code itself, there are others such as trace and metric logs that can and should be configured more globally. And for security purposes, we may want to transform our responses such as removing unnecessary headers from the client's requests or by adding metadata and sanitizing input. Transforming requests and response data between XML and JSON can be tedious and error‑prone. And unfortunately, there still are clients that need XML, so we might have to do this. And in addition to transitional authentication, some APIs may use API keys as a rate limiting mechanism or as an alternative authentication approach. While it's not incredibly common, there are still some scenarios such as server‑to‑server communication where it may be necessary. We haven't covered documentation yet, but it's a critical part of enabling your API to be consumed by clients. And depending on the function of your API, you may want to monetize its consumption. That in and of itself is a pretty involved process. Sometimes clients don't have the desire or ability to consume your REST API over, well, REST, but they would rather have a library or SDK for your API that they could import to consume the API instead. And as somebody who's used an SDK like this before, it's really nice, but keeping it up to date is a lot of work. And finally, Postman and similar REST clients have API directories or the ability to import API definitions directly for easy consumption. This is a great way to improve the usability of your API, but that too takes time and effort. Time and effort is actually how I describe this entire list of items. Yes, they're important, and yes, they're key to a successful API, but are they as important as shipping new features? Maybe not. Fortunately, all three major cloud providers have some form of API management offering. AWS's API gateway product has many, but not all of the features we're looking for, Azure's API Management product is the most developed of the three and has almost all of the features, and finally, GCP Apigee's product has a decent number of the features we're interested in. All three providers offer routing in one form or another. And similarly, they all provide authentication and authorization. And all three platforms support rate limiting in one form or another too. Caching, on the other hand, is supported by both AWS's API gateway product, as well as Azure's API Management product, but not GCP. Given it's important to a well functioning API, it's no surprise that all three providers support monitoring and logging. AWS doesn't offer the ability to systematically transform requests or responses in the same way that Azure and GCP do. Converting between XML and JSON seamlessly is a huge perk of API Management, and fortunately GCP and Azure both support it too. API keys are less popular than other authentication methods, but still used in many places. Both AWS and Azure support them. Generating a robust developer portal for documentation greatly improves the usability of your API, so it's great that both Azure and GCP support that. Similarly, both Azure and GCP support monetization workflows in their API Management products. Now of the three providers, only AWS offers SDK generation out of the box. And finally, only Azure provides a way to make your APIs discoverable in REST clients like Postman. Now, if you're like me, you probably think that these products are really cool. I mean, after all, they're offering to take over a lot of the more tedious parts of API development. However, like most cool things, they come with a cost. AWS and GCP both charge by the number of requests traveling through the API Management product. Azure, on the other hand, starts with a per‑request consumption tier and then transitions to a flat monthly rate for additional features. The biggest driver on whether or not an API management product is necessary is the API's target audience. If your consumers are primarily engineers with your own organization or you have a really great relationship with them with easy and frequent communication, you're more likely to be able to manage things on your own. Similarly, how complex is your API's access management? Do you need to configure different access for different clients beyond the scope of traditional authorization? Another significant factor is the scale of your API. If you're handling a few thousand requests an hour, you likely don't need the additional cost of one of these products. And if you're handling tens or hundreds of millions of calls a day, the expense likely makes more sense. Now as the Romans say, you have to spend money to make money. If you're planning on monetizing your API, it may be beneficial to leverage some of the tools of these products, as a positive developer experience is a significant factor in maintaining paying customers, especially in a competitive environment. And finally, how complex is your API and infrastructure? Does it contain multiple interconnected servers? Does it need to support multiple formats? Consider the feature set of your product in your selected cloud environment and factor in how much time your organization is currently spending maintaining them internally. There's a few more areas of REST that we'll cover in the next module, including the broader API ecosystem, testing, and documentation, so I hope you'll join me.

## Understanding the API Landscape

### Creating API Documentation with Swagger

In this final module, we're going to cover a number of topics that make your API more full featured, starting with the functionality every engineer hates to write, but can't live without, documentation. Now before you start to worry that we're going to have to write thousands of lines of documentation for our API to be usable, it's actually much better than that. We do have to start with comments, but they're very minimal, typically giving an endpoint descriptive name. And because REST protocols are fairly self‑explanatory, the code itself serves as documentation too. By specifying that a function is a GET or a POST, we can get a decent amount of information about that function for free. And the last piece of information we need to provide is configuration data. This usually includes where we want to generate the documentation, how we want to define models and what authentication we need and that type of information. Most popular languages have a generation tool that combines the code and the limited comments into a usable format. And the primary spec that's in use today is OpenAPI, specifically version 3.1 that came out at the beginning of 2021. We can then use the open‑source Swagger tool to consume our OpenAPI definition and create a documentation portal that describes the endpoint in a standardized way. It was actually way more exciting than clean docs because Swagger ships with a playground for you to call the endpoints in. I'm going to work through the diagram backwards by showing you Swagger first and then working our way upstream to set this up for ourselves. I've got Swagger up and running for our API, so let's dig in. We see all of the endpoints we've created so far in this course, along with some helpful descriptions of what they do. Notice that they're grouped into functionality as well. One final thing I like to point out is this lock icon at the endpoints that require authorization. Let's start by taking a look at our GET items endpoint since it doesn't require authentication and it's at the top of the list. Now we see that there's no parameters required, and we see that the response that we would get is a 200 success code, specifically an example payload. And you're probably thinking, okay, Markus, this isn't really that exciting. They're just docs. And you're right. Let's get to the real excitement. If we hit Try it out, we got to try it out. Things look pretty much the same, except now we have this Execute button here. And when we click that, it sends a real request to our API. And further down, we see that actual data coming back. But what about our endpoint that requires an actual ID? Let's check that out too down here with the getItemById. We'll hit Try it now again, and we'll set a random ID. I'm going to pick 6, and we'll execute it again. And just as before, we see our response body, as well as our response headers. At the top of the page, we have an Authorize button, which when clicked launches a dialog for us to paste in a bearer token. I've grabbed one from our authentication tester, and we're going to paste that in now. Notice how our previously unlocked icons are now all locked. Let's try our GET customers endpoint. We'll hit Try it out again, and we'll send our request, and we can see our response down below. The other thing I want to point out is that in our example curl request in the middle of the page, we see that Swagger is properly adding the authorization header with the bearer token. And further down, we see that response body with the customers. But how exactly does this get built? Let's take a look at the OpenAPI spec for our API. At the top of our definition file, we see some basic information about an API, including in the version and the root URL of the server. And then we get to the important part pretty quickly, the paths. In the get items block, we see the summary, and the responses, and the references to schemas, but what do those look like? If we scroll to the bottom of the spec, we see the items definition, including that it's an array and the definition of its children. Now while the JSON is human‑readable, we don't need to spend a ton of time here, but rather, let's see how to generate this file instead. Let's start by making changes in our code to generate the OpenAPI spec. I did preload some of the more mundane code into the before branch of the repository, so be sure to switch to that before you begin. We'll start making changes in the Swagger file inside of the config folder. Now this is the config data that I mentioned earlier. I'm using the swaggerAutogen node package to, well, auto generate the Swagger definitions. Depending on what language you choose to write your API in, you'll likely find a similar library for that language. Now remember those schemas we saw on the spec that translated into the sample data in Swagger? Here's where we're defining them as objects. This is the code I pre added for you. And further down is where we translate those objects into schemas for the spec. We haven't added any of the comments that we need to fully flesh out the spec file, so let's just see what it looks like right now. First, we'll open up a terminal to run swaggerAutogen. We will need to be in the server folder for this. And then we can run npm run swagger‑gen, and this will automatically generate the OpenAPI spec file for us. But how do we actually run Swagger and tell it to use this file? That's defined in our API's root index file. I've also preloaded this code to define our Swagger instance. In this code block, we can see the URL for our generated spec file, as well as the URL that the app should host Swagger on in the docs path. Let's run the code and check it out. A browser loads the entire route of the server, so let's slap docs to the end of the path. Now Swagger loads, but it doesn't look quite right. The endpoints aren't grouped, there's no descriptions, and we don't know what requires authentication and what doesn't. And if we expand our GET items endpoint, we see that we're missing all of the response body definitions. Let's start fixing these things. We'll start with the endpoint grouping, which we can handle in our v1 routes file. Swagger handles this functionality with a tags property, so all we need to do is add a comment to the endpoint. Now since we're registering the routers within the use function, we need to add the comments inside the parentheses. I know, I was skeptical at first too, and we'll repeat it for customers and orders too. Now if you refresh Swagger, you won't notice any changes because we need to regenerate that OpenAPI spec first. So, we'll switch to the Swagger gen terminal and rerun the command. If we refresh the browser tag, we see that our categories now show up. This is a much cleaner view for our users, especially given that we only had to add three lines of comments to do it. So let's work on labeling the individual endpoints next. In the items router, we have the functions for our various endpoints. Let's start with GET items where we'll need to add the description, as well as what the response object should look like. The description is stored in the Swagger summary field, and the response is an array where the key is the response code, in our case, 200. Then we can set it equal to an object with a description field that represents the response and the schema field that references the schemas we saw earlier in our Swagger configuration. And we'll create a similar definition for our getById endpoint. Now the POST endpoint is where things get a bit more interesting. We'll need to define both the request body, as well as a few different response codes. We'll start with the summary as normal, then specify the request body and that it's required. And next, we'll specify the schemas that clients need to match for their input. We'll then define our 201 status code along with its description and schema. And finally, we'll add an error message for our 500 error code. Now let's rerun our swagger‑gen command and reload the docs page. We now see the two GETs and POST with proper documentation. I would encourage you to take a couple of minutes to document items DELETE and PUT on your own. You can repeat this process for the order endpoints too. The fully documented code is in the after branch of this module, but what about authentication? How do we get those lock icons to show up? Let's head back to our Swagger config file. I've again added some of the boilerplate code for us so we can focus on what's the most important. I have to find an auth object that specifies that we'll be using bearer authentication. And further down, I'm taking that object and using it to define a security scheme. For our items router, we only need authentication on PUT, POST, and DELETE. Configuring security is actually really simple. We just need to add a comment and set the Swagger security property to an array with an object inside of it that specifies our security schema. This allows for multiple security methods to be supported, and we can paste that exact same line for DELETE and for PUT. But what about our customer endpoints? Well, that's even easier. We can just specify the comments at the route level just like the actual authentication. Let's run autogen and then refresh the docs. We can see now that our items are properly documented. If you haven't already, I'd encourage you to finish the customer and order endpoints to practice. It's fully implemented in the after Git branch of this module. Now you've noticed that I've got this version selector, but it only has v1 in it. I've included the code on how to add an additional version in the after branch, so definitely check that out. It's more or less the same process that we just did, but with a new OpenAPI spec file. Now Swagger is great for allowing your users to test your API, but what about testing that you might want to do? We'll cover that next.

### Testing Your API

So far, the testing we've done is what I would broadly consider confirmation testing. We wrote some code, confirmed that it did what expected. Sometimes we made sure other things didn't break, but broadly, our testing efforts have been pretty minimal. Let's talk about more robust testing that we should be doing. The simplest check is for reliability or uptime. Essentially, is your API up when clients expect it to be up? There are literally thousands of uptime monitoring platforms out there, and it's easy to understand why. Uptime monitoring essentially comes down to the tool making a REST request to the endpoint you provide and then recording whether or not it's successful. If it's not, then you get an alert. Validation testing is a fancy way of saying did the code do what you wanted it to do? Typically, the bulk of this testing is done as unit tests directly within the code, but it is possible to do end‑to‑end testing at the API level too. I typically see this occurring in non‑production environments where there's a stable set of data to test against. Load testing used to be a substantial focus of many organizations. And while it's still important, it has become less prevalent as cloud computing and autoscaling resources becomes more commonplace. In essence, the question load testing answers is how many concurrent connections can your API support before it begins to degrade? And finally, we have security testing. While this can be manual, many organizations are now relying on automated security testing tools that check APIs for known weaknesses. Let's take a look at Postman, an incredibly popular REST tool that addresses several of our testing needs. If you haven't heard of Postman before, it's a full‑featured API client primarily available for desktop. Because we generated an OpenAPI 3 definition of our API previously, I was able to easily import it into our Postman collection. Now I've also set a number of variables for this collection, including the server URL and token that we'll get from the authentication tester. If you want to follow along, I've included this collection in the root of the Git repo for you to import into Postman, if you wish. If we take a look at one of our endpoints, everything looks fundamentally the same as Thunder Client, the REST client we've been using in VS Code. But we can use Postman for functional testing. I'm going to write a simple test here to confirm that our response contains hiking boots. There are a ton of snippets to choose from to get you started. When we send the request again, the test executes, and we see its status below. Postman also has a bunch of testing templates to choose from that are based on common industry scenarios. These tests can be scripted to pass data from one endpoint to another. For example, we might want to get a list of orders, take one of the order IDs, and then get its details. We can also do some limited load testing in Postman. On our Collections tab, we see the Run button, which opens up the runner. On the left, we have the full list of endpoints in the collection. I'm not going to performance test all of them, but rather just to get all items called. Then I'll pick the performance option and create a simulated peak load of 100 users over a minute. Once the test starts to run, we see the users represented by the gray, the average response time in blue, and the error rate in red. Now 100 concurrent users isn't that much, but it's a good place to start. And the paid Postman plan supports a higher number of virtual users. While Postman is a great tool for load and validation testing, there are certainly other options as well. One of the original and still prominent contenders is JMeter, a Java‑based desktop application. Now JMeter supports all types of web testing, including UI and REST, making it very versatile. It also has a really steep learning curve, and it can be challenging to scale. Taurus is an interesting tool, as it is a Python library that allows you to write JMeter tests as YAML files. Personally, creating an additional layer on top of a different testing language like this often leads to problems. And if you want a feature that's implemented in the base language, but that your library hasn't implemented yet, you have to wait. Regardless, Taurus is an up‑and‑coming approach for many API testers. And finally, there are a large number of cloud‑based testing solutions. Typically, they fall into two options. Many allow you to upload JMeter or Taurus tests that you've written locally, and then they'll execute those for you, often on a larger scale than you can support locally, and report the outcomes. Alternatively, they may provide some more rudimentary support, including which endpoints you want to call with which data and how many times. Honestly, either option may be appropriate, depending on your use case. Now there's a ton of API security testing tools available, all with different purposes and price points. Rather than going through the massive list, I want to highlight the three main types of security testing to help you figure out what tooling you might need. First is security posture, which are essentially API cataloging tools. The goal is to have a full understanding of all the API endpoints of an organization, as well as the data they interact with. This is incredibly important to ensure that there aren't any unmonitored or poorly secured endpoints. Next is runtime security, which as the name implies is all about monitoring APIs while they're running. These tools look for unusual usage patterns and make sure that APIs are being used appropriately. And finally, the security testing tools, which are more like vulnerability scanning. The goal here is to identify vulnerabilities in the APIs before they can be exploited. I've typically seen security teams run these tests before a new or significantly modified API is released to the internet. And as one of my favorite security engineers would always say, there is no test internet, meaning we need to get our vulnerabilities in check before we release the code to the world. If you're looking for a great list of security testing tools, check out the API security tools page on OWASP. I've included the link at the bottom of the screen. Now in the next clip, we'll take a look at how our API fits into the larger ecosystem of API directories. I hope you'll join me.

### Understanding API Directories

So far, we've focused exclusively on data that Carved Rock maintains internally such as their inventory, customers, and orders. Let's explore how to leverage external data sources through API directories. Here are a few scenarios Carved Rock is considering and would like to leverage external data for. As an outdoor retailer, potential buyers purchasing trends are heavily influenced by the weather. Company leaders are hoping to use a weather API to predict when to stock more snow gear versus warm weather options. This is a great opportunity to leverage APIs that are available to the general public, often at a cost. You may have noticed that our items dataset was pretty bare bones. Carved Rock wants to leverage the extensive product information from some of their vendors to provide customers with a more immersive experience. It's fairly common to leverage APIs with your organization's existing partnerships. In fact, in some of my past roles, I often prioritize API availability and quality during vendor selection processes. And finally, there may be scenarios where your data is sent to an API rather than being pulled. Typically, you would set up a POST endpoint, commonly referred to as a webhook receiver. You would configure this in the incoming API's webhook settings. This would allow you to configure the data you receive and at what frequency. Carved Rock is interested in leveraging an inbound API to collect information about influencers who earn commission by encouraging their followers to purchase products from Carved Rock. Before we move on, I'd encourage you to pause the video and think about what types of data your API could use in each of these three categories. Finding the perfect API for your use case can be tricky. I've gone on many fishing expeditions myself, and trying to find an API that checks all the boxes can be quite the challenge. One of the best places to start is an API directory, which is, as the name implies, a list of APIs. Now there's plenty to choose from, but right now, apis.guru is quite popular. In addition to robust search, it also validates the API definition files and converts them to OpenAPI 3.0. Rapid API's API hub takes things a step further. You can't browse their participating APIs without creating an account, but there are additional features once you sign up. We'll take a deeper dive in just a bit. Now of course, there's nothing wrong with just seeking out potential APIs on your own. Oftentimes, my fellow engineers have had great recommendations for APIs they've used previously. And if all else fails, just Google the problem you're trying to solve and add API to the end of your search. It's always a great place to start. Let's take a quick dive into the Rapid API hub. So here we see the search results for whether to match Carved Rock's use case. One of the biggest benefits is the Rapid API calling these APIs on a regular basis to report their average response time, as well as uptime. Let's take a look at Open Weather. We see the required parameters, as well as a code snippet to use if we want to try it out locally. This is all pretty powerful, especially if you're looking at multiple APIs at once. I do want to call out that Rapid API is taking care of the authentication and billing for all of the APIs in its directory. So rather than having to manage authentication for every API you consume, you can just manage your Rapid API key in your code. Let's talk about vendor partnerships. I'm broadly thinking of partnerships that extend or exist beyond the needs of your API. For Carved Rock, it might be a supplier of kayaks or the manufacturer of their cash registers. To the extent that your influence allows, I would encourage you to push for API access during these contract negotiations. It's always easier to ask for APIs upfront, then hope that the vendor will provide them at a later date. Of course, one of the top concerns in this process is what will the data actually look like? What objects and fields will you be provided with and to what extent will you be able to customize requests? In other words, what's the shape of the data? How regularly will this data be updated? I once had a vendor try to convince me that they had an API because they could drop a CSV file onto a file share for us once a week. I was pretty unimpressed by that option. Along those lines, will the data be in a format you want to work with? More often than not, if JSON isn't available, it's a potential red flag. You also want to feel supported in your efforts to work with this API. One of the best places is to get a feel for what your experience is going to be like is to check out the documentation. I was evaluating a vendor once who claimed they had a very robust API. When I asked for the docs, I received a PDF file that was several years old. We didn't move forward with them. Similarly, how invested are they in helping you get started? Great vendors will offer meetings or other support to help you get the API up and running. It's also really important for you to understand how much influence you have over suggesting new features. If your requests are never answered, it makes it really hard to make forward progress. I once worked with a vendor that had an onsite conference I attended. I was expecting a sales pitch, but they paired me with one of the developers working on the features of the API I was using. Together, we changed their API to add the functionality I needed in near real time. It was incredible! And finally, let's talk about how to handle inbound APIs. First and foremost, security is a top priority. OWASP highlighted giving too much trust to partner APIs compared to user data as a security top concern. With that in mind, it's important to understand what the authentication methods of the inbound API are available. The OAuth 2 workflow we implemented earlier is likely off the table for many APIs. However, a weaker authentication option can still be strengthened by restricting the source IP addresses that the requests are originating from. Inbound API providers should be able to supply this information to you and update it as it changes. This will allow you to block requests to your endpoints from all other IP addresses. And finally, it's important to understand exactly what the inbound API will be sending over and what format it will be in. Then, you can create validation that aggressively enforces this format. But what about making your APIs publicly accessible? I'll be honest. This is going to be a lot of work and could easily be a course on its own. Let's go over a few key considerations. I'm a big fan of user‑centric approaches, and this is no different. Start with who will be consuming your API. Consumers, small or medium business, enterprise customers, government agency, the list goes on and on with each type of consumer having their own unique challenges. Some of those challenges are their specific expectations. Remember all of those items I just highlighted in assessing vendor partnerships? You should expect that potential customers of your API have those expectations of you too. Security, of course, is a top concern for you, but it is for your customers too. And depending on what industry you target, you may face specific security requirements. Both healthcare and finance in the United States have significantly more complex security requirements than other industries. Similarly, some organizations will have a stricter security posture than others. For example, it's not uncommon for large organizations to be very prescriptive on their security requirements, including technical details such as defining the acceptable validity period for OAuth tokens. And finally, consider if you want to generate revenue from your API. If you do, consider what the pricing structure will look like. Will you charge per request, a flat monthly fee, some combination of the two? Similarly, how are you hoping to get paid? While requiring a credit card for subscriptions is fine for individual consumers and small and medium businesses, most large organizations prefer pairing by invoices instead. A good rule of thumb is the larger the organization, the harder it is to find someone with a company credit card that's authorized to pay the bill. This has been quite a bit of information, so I'd encourage you to take a break to digest it if you need. In the next clip, we'll be taking a look at some REST alternatives, so I hope you join me on that journey too.

### Exploring REST Alternatives

I mentioned previously that REST is the biggest API communication protocol in the world. And while that's true, it doesn't mean it's the only one. The most prominent competitor to REST is GraphQL, whose first stable release was in 2021. The best way to understand GraphQL is to compare it to some of the main points of REST. REST has a fairly rigid data structure. Each endpoint has a defined set of data it returns. And while you can use a query parameter to modify the volume of data you get, partly it's pretty fixed. GraphQL, on the other hand, supports a flexible structure by allowing clients to select which fields they want to get back from an object or list of objects. We're also used to REST having an endpoint per object and operation. GraphQL has a single endpoint for the entire API, and the data returned is controlled entirely by what the client requests in the body. REST's simple structure really shines when the relationships between objects are minimal. When they get more complex, you have to make more and more API calls and combine data on the client side. This is one of GraphQL's biggest strengths, as it allows you to pull data and relationships in a single call. In a way, GraphQL essentially front loads all of the data arranging work onto the server teams instead of the clients. Ultimately, that relationship between objects still has to be defined somewhere. Now clients can define their queries in GraphQL because the data and relationships are strongly typed, and this typing is shared publicly with them. This adds some complexity compared to REST, and it takes almost no set‑up time to begin consuming REST APIs. And while you can use a bare bones library for GraphQL, you lose many of the benefits that GraphQL offers like strong typing. And I'm going to let you in on a little secret about GraphQL. Under the libraries and all the validation and everything else, it's still just a bunch of POST requests. So let's do some side‑by‑side comparisons between GraphQL and a REST endpoint. We'll get started with a list of all of the items with an extra image and review data. The REST example probably looks somewhat familiar. We've added an include query parameter and specified images and reviews. Notice that I'm sending a GET request for our items endpoint. For GraphQL, we'll send a POST request to our single endpoint named graph, and then the body will specify our query, the item's collection, and then passing the specific fields we want in our dataset. Let's take a look at another example that highlights GraphQL's power a bit better. Let's say we want to get a list of all of the customer's order along with the details of the items in those orders. With REST, we'd make a call to our customers endpoint, which fortunately already includes a specific version for a list of all of the orders. Then we'd have to call the orders endpoint for each of the order IDs from the previous call. Now fortunately, all of our orders endpoints include a small item object that has all the data we need. But if it didn't, we would need to call our items endpoint with each itemId from the orders with our previous calls. And finally, we would assemble all of this data into a single object for us to use. Now in reality, the clients would raise this need to the API team, and they would hopefully create a query to include all of the right parameters. In GraphQL, we'd start with our POST request to the graph endpoint again. In the body, we'd pass the query, this time specifying the customer collection and include a customer ID as a parameter. More on that in just a second. Then within the customer, we'd specify the order collection along with the orderId property. Next, the item collection, which is where we'd specify the ID and name property, along with closing the query above. Then we'd specify a second top‑level object called variables, which is where we pass in the customerId. And this query would return a single object with the data formatted in the way that the client specified. And this is great for the client, but I do want to call out that all this assembling still needs to happen somewhere. And in GraphQL, we're just offloading that work to the server team instead of the clients. The other modern approach to REST is called gRPC, or g remote procedure calls. What does the g in gRPC stand for? Great question. I actually haven't been able to find an answer yet. I suspect it might be Google since they created and continue to maintain it, but that's just a theory. So gRPC is a specific message exchange format that's designed to move a lot of data very quickly. And it does this by leveraging custom protocol buffers to transfer data rather than relying on HTTP protocols. Because of this, gRPC is limited almost exclusively to server‑to‑server communication. It is possible to build a proxy server that converts gRPC traffic to HTTP traffic for the client to consume. A significant barrier is that the tooling is pretty limited for gRPC. There are implementation libraries for the basic functionality in several popular languages, but it's nothing compared to the vast ecosystem oppressed. And as a result of these challenges, adoption of gRPC remains very low with only a few major companies leveraging it. I've covered the major modern APIs, but what about some not so modern ones? Let's start with SOAP, the dominant API solution before REST. SOAP has a strict data contract between servers and clients that's XML‑based. The contract is very, very verbose, and data transfers are slow and really heavy. I've consumed SOAP services at the beginning of my career, and it was rough. Most of the responses we received, no matter how small or powerful the hardware, we still had response times in the seconds. SOAP is also really, really hard to implement. The team that made the API that I consumed in the past literally had to make an instructional video explaining how to configure the equivalent of a Hello World app for SOAP. The video was over 45 minutes long. Not to mention the shared contracts were so difficult to consume that we ended up writing a helper library to interact with the SOAP contract library. Overall, it was considered to be slow and ineffective in the early 20 teens, and it has not aged gracefully. I hesitate to call this next approach an API, but technically data is being shared between two servers. In this approach, one party sends static files to another party's FTP server. These files are in a previously arranged file format. Now whenever I've worked in these in the past, we hope for XML, but usually ended up with some sort of proprietary format that the sending party made up to meet their own needs and we had to deal with. This approach is fairly common in the insurance and finance industry, and its primary use case is for data that isn't real time. Think of an upload once a night or once a week, for example, I will say that this is an effective approach to upload a lot of data really quickly. Ultimately, once you ingest the file and confirm that it's safe, it's typically just a matter of looping through each of the records and performing a database create for each one of them. We've talked about REST and its alternative, and hopefully I've sold you on REST. In the next clip, I'll share how to share your excitement with others.

### Advocating for REST

We've talked about the technical in and outs of REST, and I think you're in a good position to handle many of the scenarios your organization might face in this space. But how do you convince others to give REST a chance? One of the best ways I found is answering the common question, what's in it for me? This is a great time to highlight some of the biggest benefits of REST. While everybody else is doing it typically isn't a great reason to do something, it can be in technology. Because development is so interconnected, being the industry standard for communication is a huge benefit of REST. And at the beginning of the course, we set up a REST API with just a couple of handfuls of code. While that was a JavaScript‑specific example, most languages let you get started on REST really easily. And unlike other protocols, REST allows you to communicate with any type of client or server. You're not restricted by hardware limitations. And speaking of limitations, I think you'll be hard pressed to find a modern programming language that doesn't support REST. There's actually a huge selling point for REST. You likely won't need to change the language you're working with already since it should support REST. Especially if you're able to implement your API as stateless, REST is infinitely scalable. By not requiring a specific connection between a client and a particular server, adjusting servers to handle changing demand is seamless. And finally, REST is just simple. There are some conventions around naming, but beyond that, it's an incredibly flexible protocol. Okay, so we've got you some answers for what's in it for me, but let's talk about how you might implement REST in your organization. If you already have existing functionality that could be replaced by REST, assess its current state, including what language it's written in, how supported it is, and the organizational desire to replace it. From a non‑technical standpoint, I always recommend identifying key stakeholders, as well as people who will help you promote your ideas to others. Just as we did in the course, I encourage you to set up a REST playground for people to interact with. Specifically, it's really important that this playground does not have access to your production code or data. Then, I'd encourage you to go through the process of creating a small feature. Ideally, it's something that gets a decent amount of use, but isn't part of the critical path. Once you've built it locally, start working on getting it to production. This is a great time to think about some of those API Management topics we discussed earlier. Then repeat these last two steps as you convert more and more of your existing functionality to REST and add more new features. And finally, I want to call out that it's really important that you continuously evaluate the implementation, and that's not just from a technical standpoint either. Are your peers comfortable writing REST code? Are other teams like security and DevOps okay with how the rollout is going? Take a couple of minutes and brainstorm what this process might look like for you and your organization.