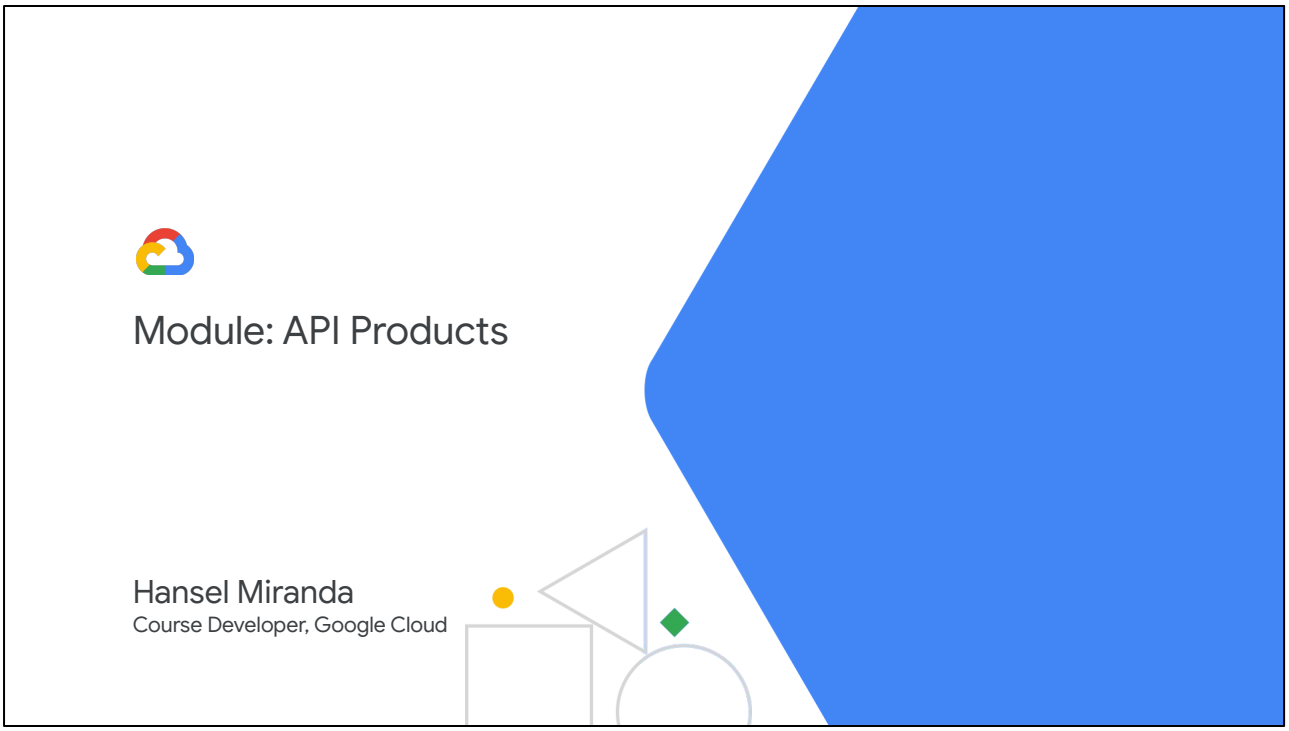



In this module, we'll learn about API products.

API products are used to package APIs for use by app developers in their applications.

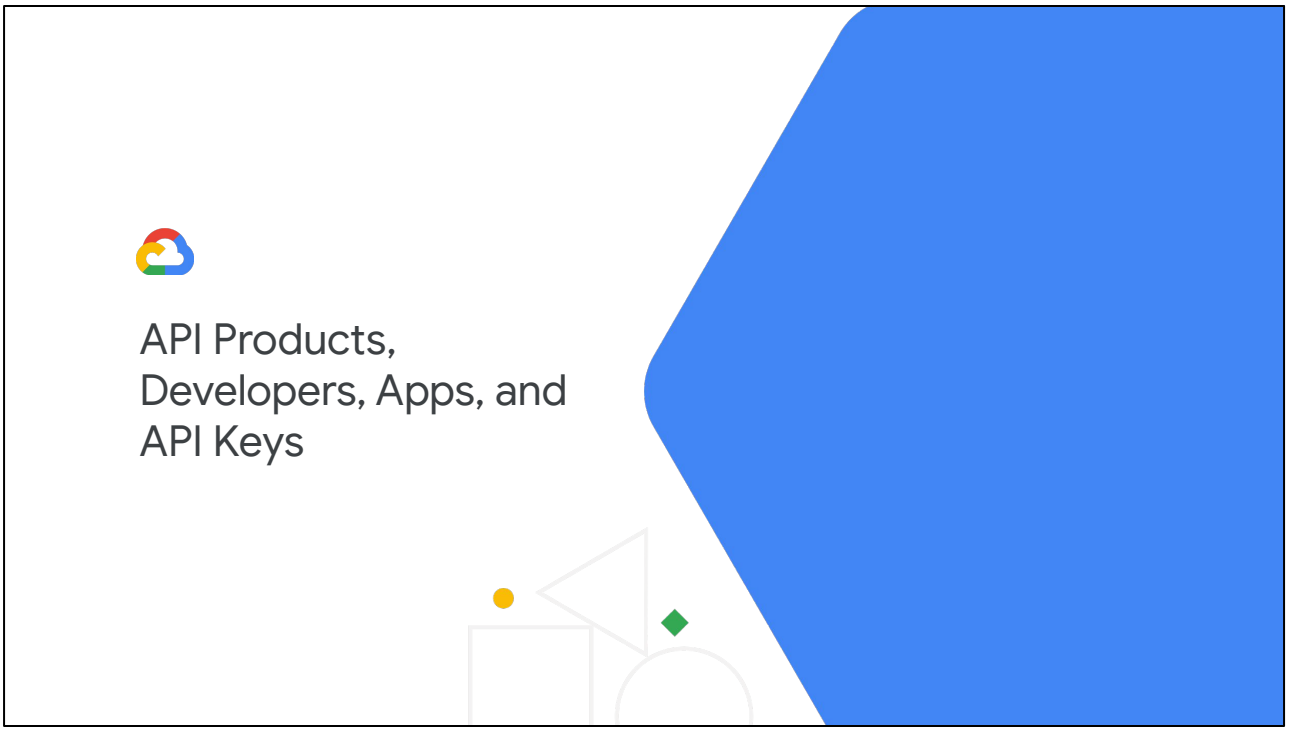
You'll learn about how API keys can be used to protect your API from unauthorized access.

You will also learn about API product strategies, and we'll use an example scenario showing how API products can evolve over time.

And you will learn about REST API responses and how you should use HTTP status codes in your REST APIs.

In a lab that adds API key protection to your retail API proxy, you'll create an app developer and an app, and call the retail API using the app's credentials.

Another lab will show you how you can control the functionality of an API proxy using custom attributes attached to an API product.




In this lecture, you will learn about four entities that are vital to providing controlled access to APIs on Apigee: API products, app developers, apps, and API keys.

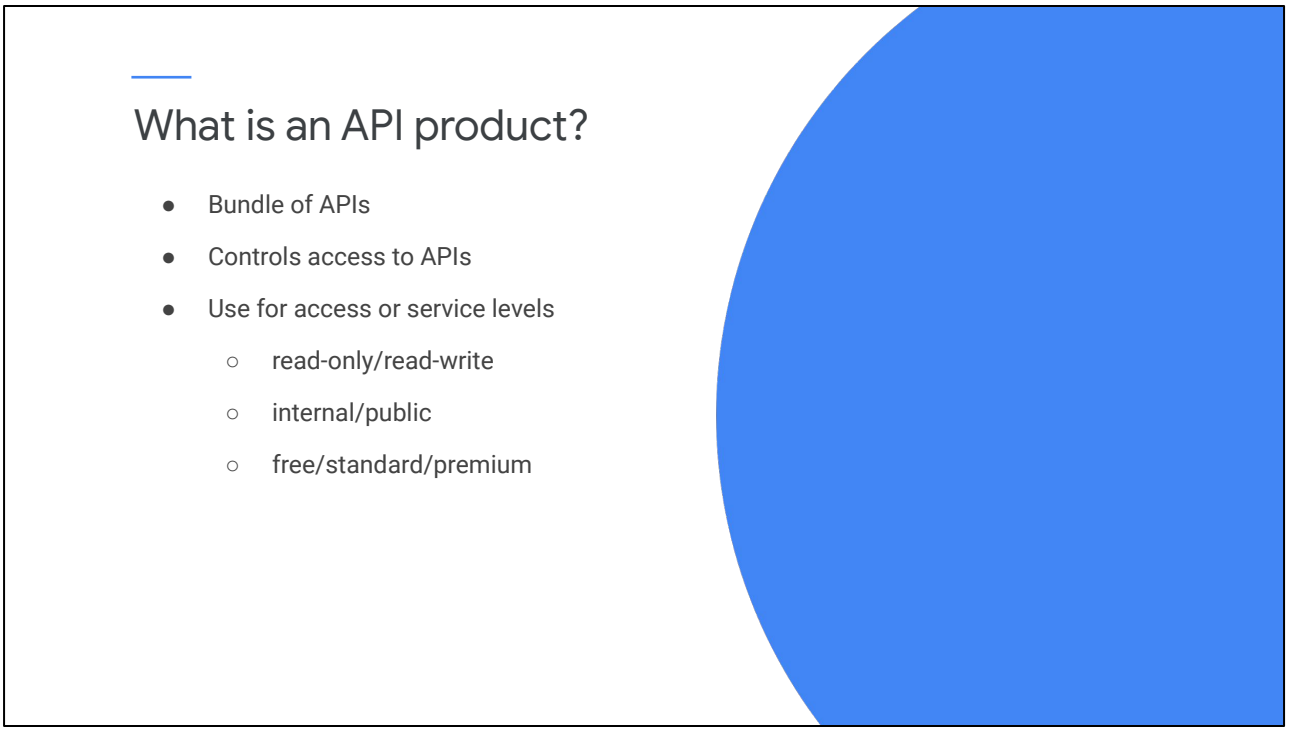
We've already learned about API proxies, which are used to implement the APIs that are used by app developers within their apps.

You've heard about developers and apps.

API products are how we provide API access to our apps.

And API keys are used by apps to gain access to the APIs.

Let's start with API products.




What is an API product? Stated another way: in the context of APIs, what is a product?

In Apigee, an API product is a bundle of APIs. These APIs are typically packaged together with the needs of application developers and applications in mind.

For example, we might create an API Product for selling items online. This API product might contain several APIs, including product catalog, shopping cart, and shipping APIs. We can decide to document and market these APIs together as a bundle, as an API product.

We also use API products to control access to our APIs. We may want to block public access to our APIs, or track the apps using our APIs. We can grant access to our APIs at the API product level.

We might also want to allow different apps to access the same selling APIs, but with different levels of access, or different levels of service. API products are a great way to do this. This can be as simple as allowing read-only or read-write access to the same APIs.

Another example is internal versus public: a company's internal apps are usually given more access to sensitive data than apps developed for the public. Or maybe free versus standard versus premium.

We could allow free use of our APIs, but provide better features and more capacity for standard and premium customers.

We will look at more API product design strategies in the next lecture.

## API Product configuration: Custom attributes

- A variable for attribute name and value is populated when the app is identified using an API key or token *(details later)*.
- Populated variables are used to dynamically control the functionality of the API proxy.

Examples:

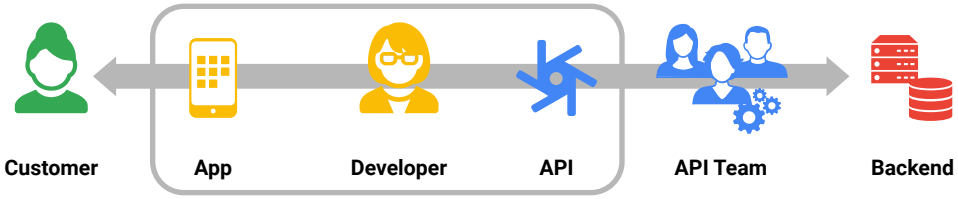
| | | | |
|---|---|---|---|
| accessType: | internal/external | batch-size: | 10/100 |
| service-plan: | free/standard/premium | requests-per-sec: | 1/10/100 |
| readOnly: | yes/no | max-speed: | low/medium/high |

When you're designing API products, one important feature is custom attributes.

An API product custom attribute is a name/value pair that will automatically be populated as a variable when an app makes a request.

This allows your proxies to dynamically grant access and service levels that are appropriate for each app, based on custom attributes of the API product associated with the app.

## Digital value chain



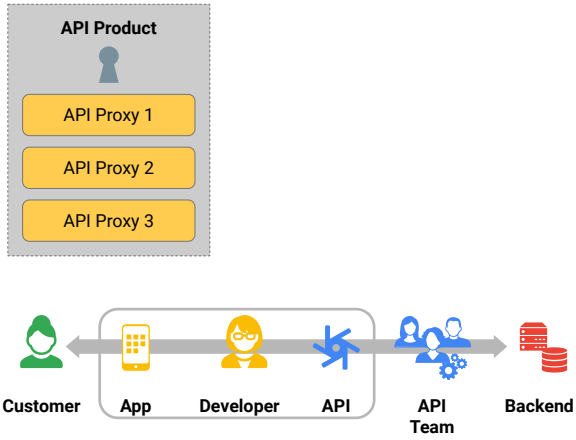

Remember the digital value chain?

APIs are published for use by app developers.

App developers build the apps that create the connected experiences that will engage customers using the apps.

## Publishing APIs

- APIs are packaged in an API Product.

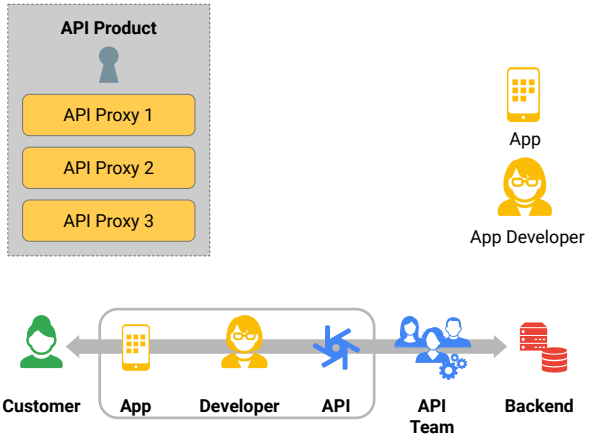



So how do we publish APIs on the Apigee platform?

The first step is to package APIs as API products so they can be consumed by app developers.

## Publishing APIs

- APIs are packaged in an API Product.
- App developers create apps that consume APIs.

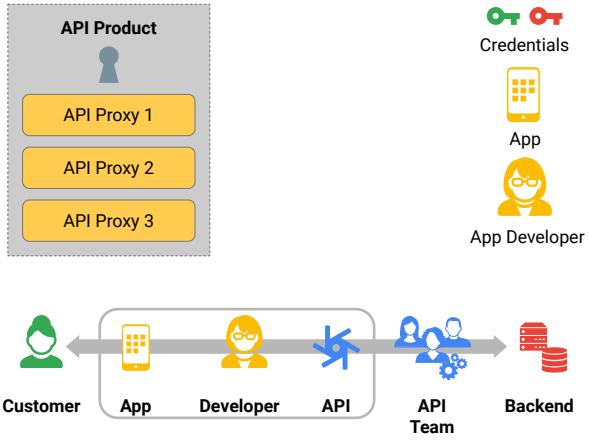



App developers typically sign up on the API provider's developer portal to get access to the APIs. Here they can explore the APIs, bundled in API products.

App developers can register their apps to use API products, which gives them access to the corresponding APIs.

## Publishing APIs

- APIs are packaged in an API Product.
- App developers create apps that consume APIs.
- Credentials (consumer key and secret) are created for an app and associated with one or more products.



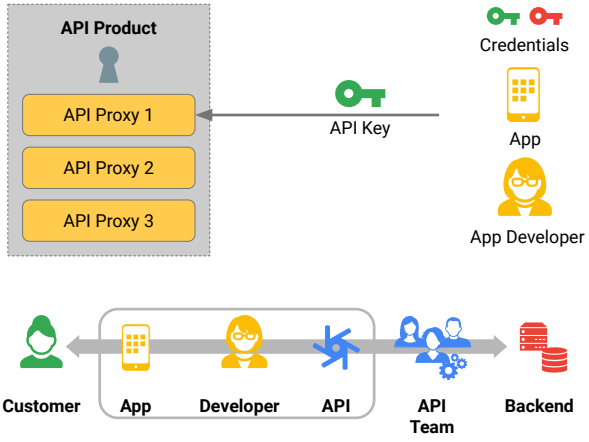

Apps are given credentials that are associated with one or more API products.

The credentials are called the consumer key and consumer secret.

These credentials are unique to the app.

## Publishing APIs

- APIs are packaged in an API Product.
- App developers create apps that consume APIs.
- Credentials (consumer key and secret) are created for an app and associated with one or more products.
- A consumer key can be used as an API key to access protected APIs.



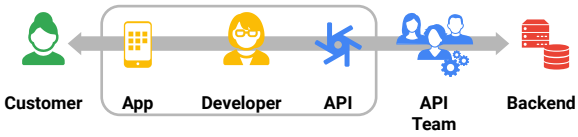

The consumer key can be used as an API key.

The API key is sent with requests to the API.

Requests without a valid API key that has access to the proxy can be rejected.

## App developers

- App developers are configured with:
  - Name
  - Email address
  - Username
- Custom attributes can be attached to an app developer.
  - Attributes are available as variables in an API proxy.

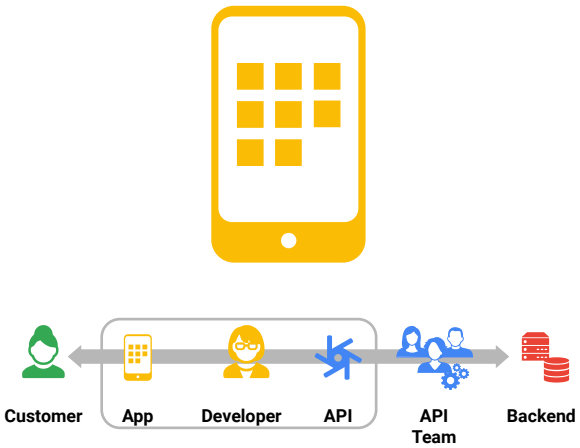



In Apigee, app developers are configured with a name, email address, and username.

Custom attributes can be attached to a developer, and these attributes will be available as variables in the proxy when the API key is verified.

## Apps

- An app is associated with an app developer.
- Custom attributes can also be attached to an app.
  - Available as variables in the proxy
- One or more sets of credentials (consumer key and secret) can be created for an app.
- Credentials may be associated with one or more API products.
- Credentials can be revoked.




Apps are created for an app developer.

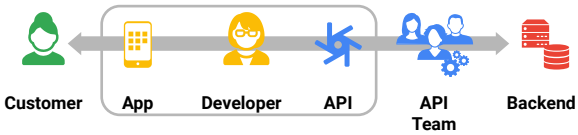
Like app developers and API products, apps can have custom attributes. The attributes are available as variables in your proxy.

One or more sets of credentials can be created and attached to an app. The credentials, a consumer key and consumer secret, can be associated with one or more API products.

The credentials can also be deleted by the app developer or revoked by the API team, which removes access to the associated APIs.

## API key

- A consumer key is used as an API key.
- Allows access to APIs protected by VerifyAPIKey policy.
- If API key is valid, variables are created for associated app, developer, and API product attributes, including any custom attributes.




The consumer key can be sent with an API request, acting as an API key that identifies the app.

The VerifyAPIKey policy validates the key and allows or denies access.

If access is allowed, any custom variables configured for the API product, application, and developer will be populated.

Note that if your credentials are associated with more than one API product, the attributes for only one of the API products will be populated.

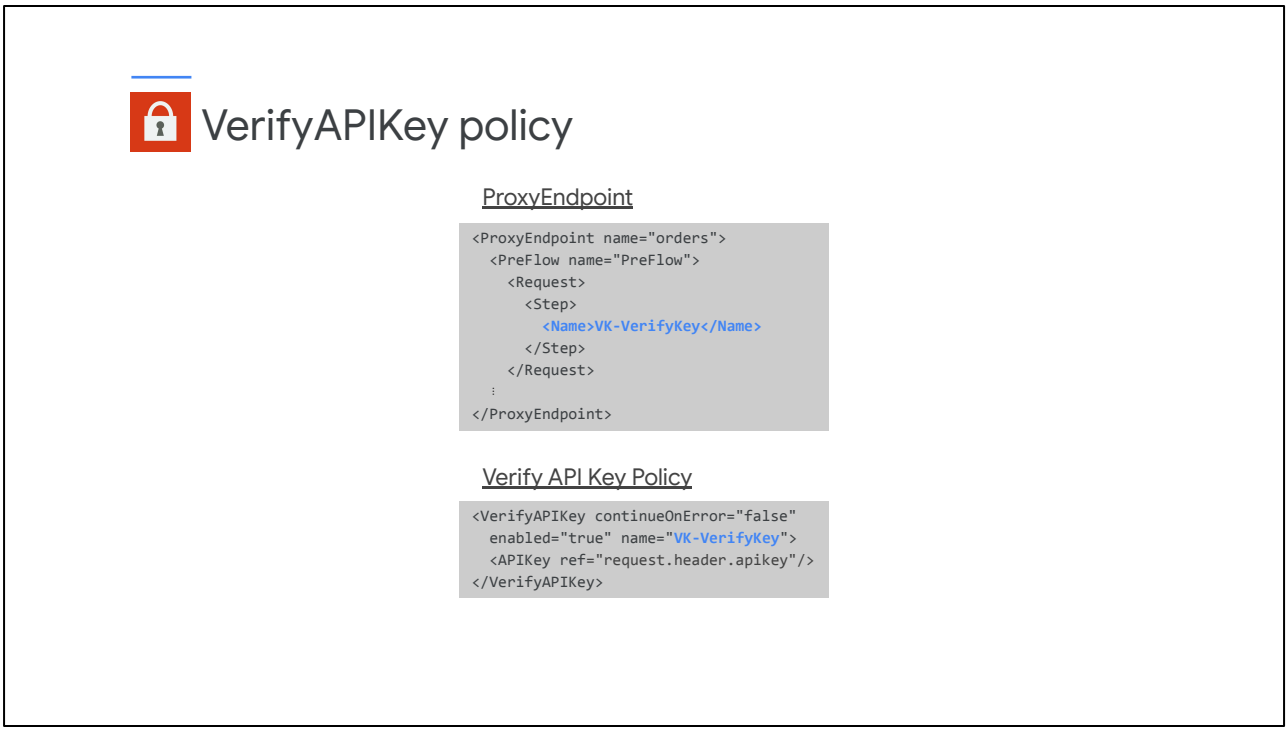
# VerifyAPIKey policy

ProxyEndpoint

```
<ProxyEndpoint name="orders">
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>VK-VerifyKey</Name>
      </Step>
    </Request>
  ⋮
</ProxyEndpoint>
```

The VerifyAPIKey policy is almost always attached in the proxy endpoint request preflow. Verification of the API key should occur very early in the proxy processing.

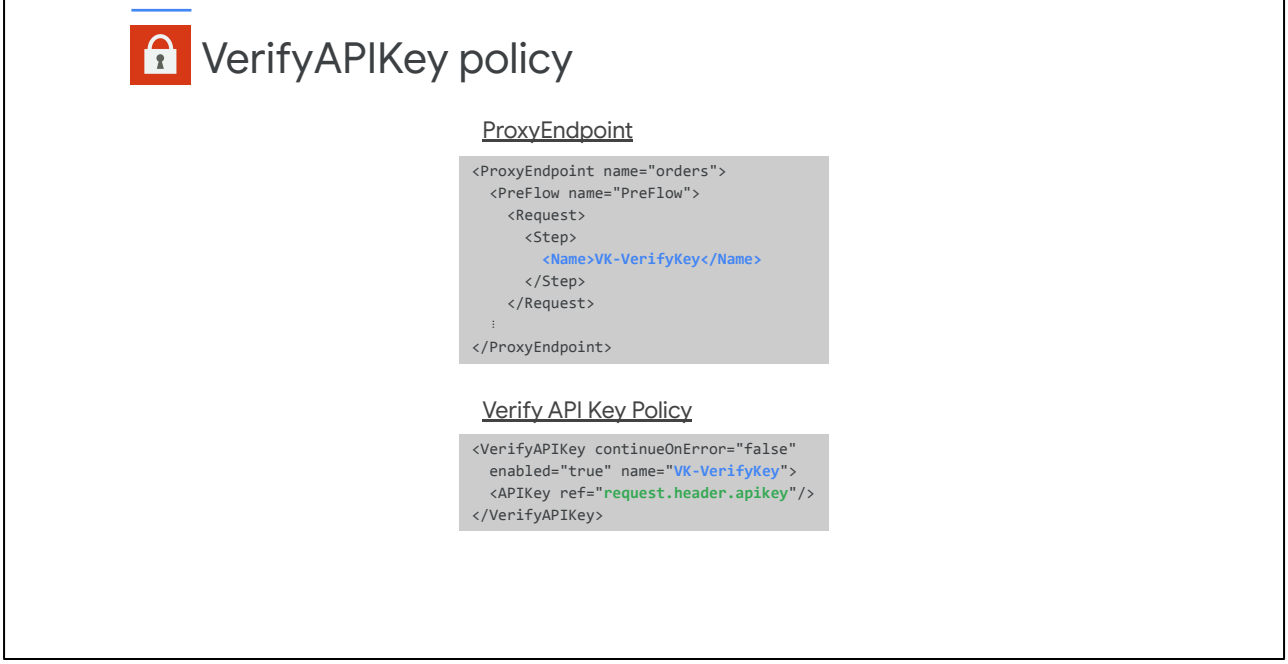
If the caller does not send in an API key, or the key is not valid, the policy will reject the request.




This is the XML configuration for the verify api key policy. Policies are always defined in their own XML files.

The name field in the XML root element specifies the policy name.

The policy name, VK-Verify-Key, is specified in the step in the proxy endpoint preflow code.



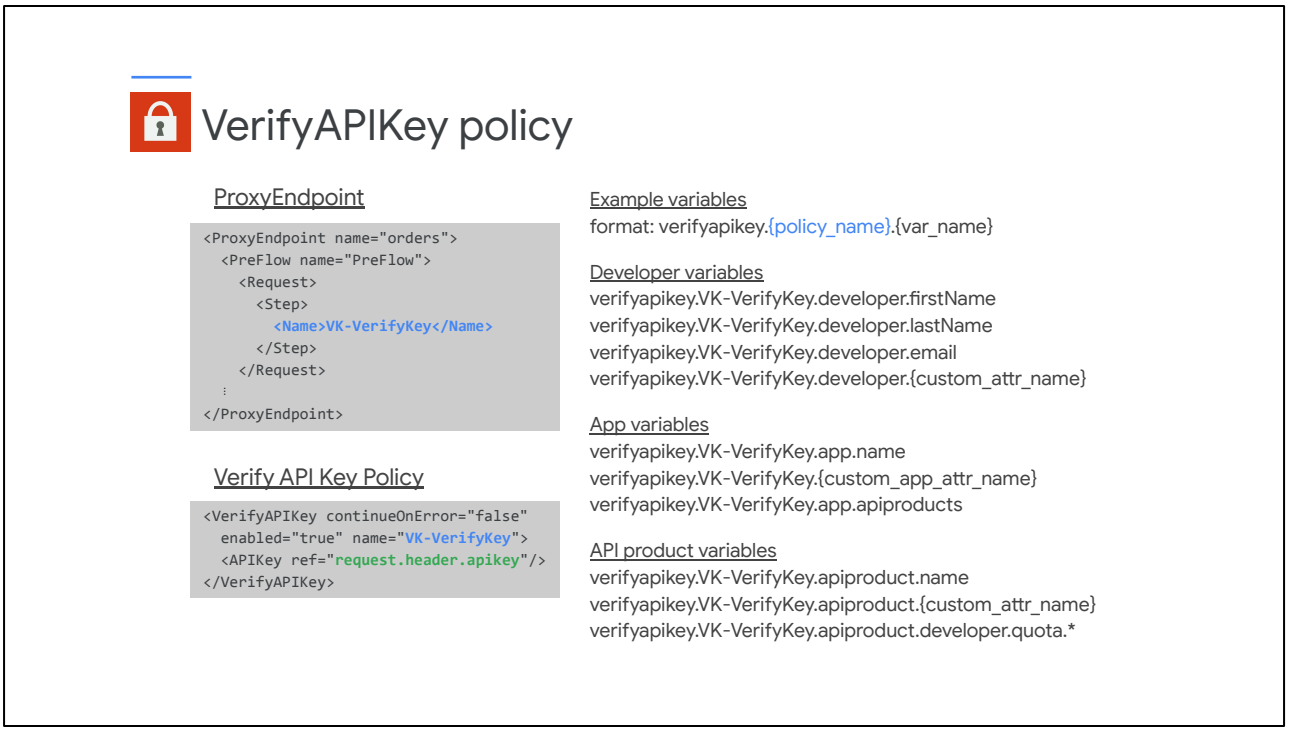

Most policies have several configuration elements, but a VerifyAPIKey policy will typically only have the API key element.

The API key element specifies the variable where the API key is found. By default, the VerifyAPIKey policy looks for the API key in a query parameter named API key. This example is configured to look for the API key in an HTTP header named API key.

Note that HTTP headers are case-insensitive, unlike query parameters, so any capitalization of API key will work.

When this policy is being executed, the variable request.header.apikey will be checked for a valid API key. If the API key does not exist, or is not valid, or the associated API product does not allow access to this API, the verify API key policy will raise a fault. Raising a fault causes processing to stop and an error message to be returned.

We will learn more about faults later.

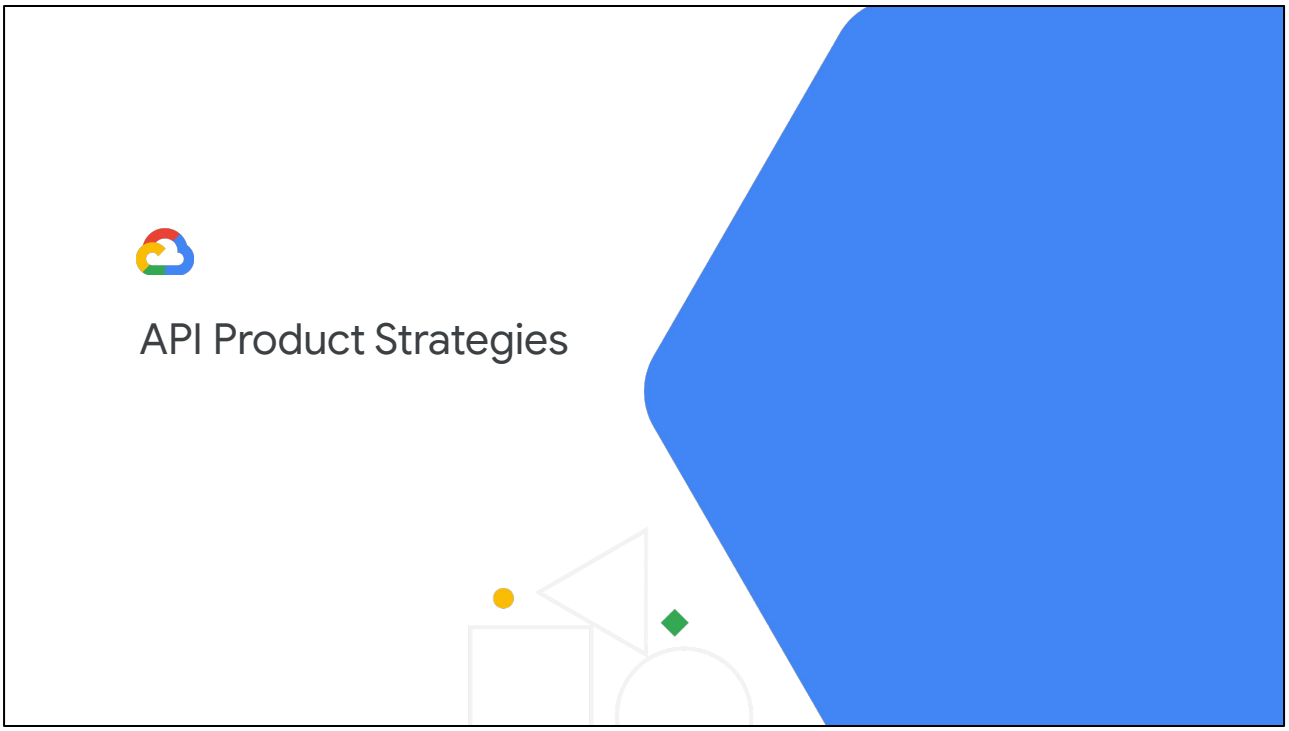



If the API key is valid, variables will be created and processing will continue with the next step.

Variables are formatted like this:

- The first segment, VerifyAPIKey, is the type of policy.
- The second segment is the name of the policy. In this case, the name is VK-VerifyKey.
- The remaining text is the name of the variable or custom attribute.

You can see here some examples of the variables that will automatically be populated by the VerifyAPIKey policy.




You've learned the basics of API products and how they are used to package APIs.

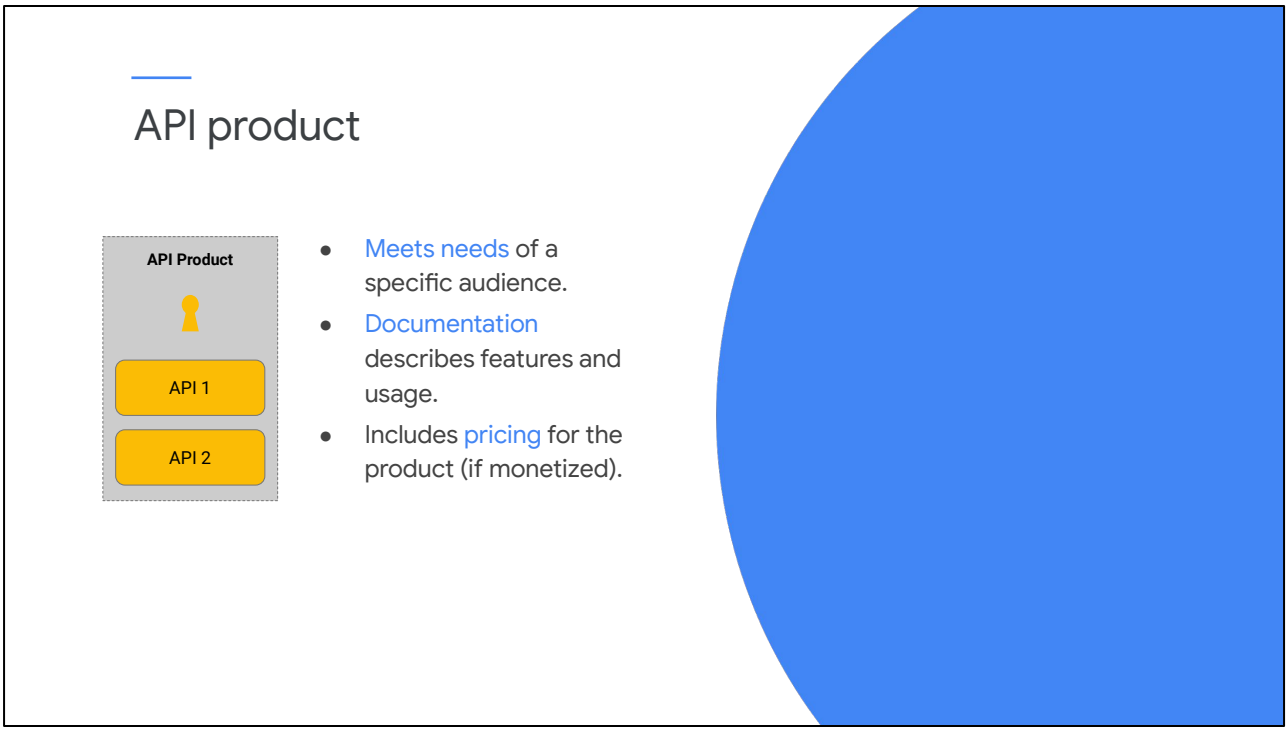
Let's explore the strategies you can use to design your API products.


Before we discuss API products, let's look at some characteristics of a traditional product.

First, a product is a good or service that attempts to meet the needs of a specific audience. If there isn't anyone who wants the product, there isn't much point in creating it.

Second, products typically have documentation explaining the features of the product and how to use it. Documentation is important for helping the consumer get the most out of the product.

Third, unless you are giving the product away for free, products have a price. Some products provide value to a company in other ways, so the product's price might be subsidized, or free, to make it more attractive to a consumer.




OK, so let's look at API products. What are some characteristics of an API product?

It turns out that API products are very similar to a traditional product.

Like a traditional product, an API product attempts to meet the needs of a specific audience: the app developers who will use the product.

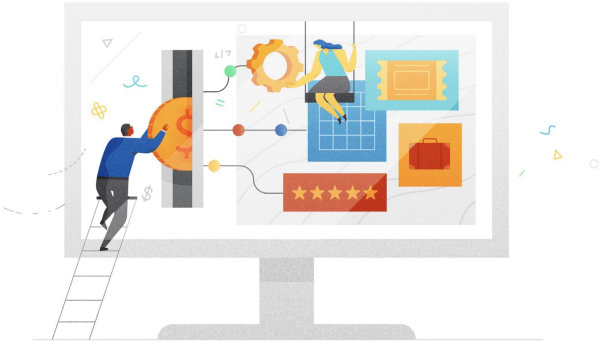
Documentation explaining the features and usage of your APIs is also important. We use OpenAPI specifications to allow app developers to discover and try out our APIs. Great API documentation can result in higher developer satisfaction.

In some cases we might want to charge for our APIs. Many APIs drive revenue directly or indirectly just by being used, so APIs are often provided at no cost to app developers.

However, we can choose to charge for our APIs if we are providing a valuable service to the app developer. Or we might even provide revenue sharing, by paying developers a percentage when they sell our products through their apps.

If you want to charge or pay app developers, API products and Apigee API monetization can help you do that.

## Apigee API Monetization



- Monetize APIs to generate revenue for API use or pay app developers for revenue sharing.
- Sign up, set up billing, process credit card payments, and access monetization reports in the Developer Portal.
- Available for Enterprise offerings.

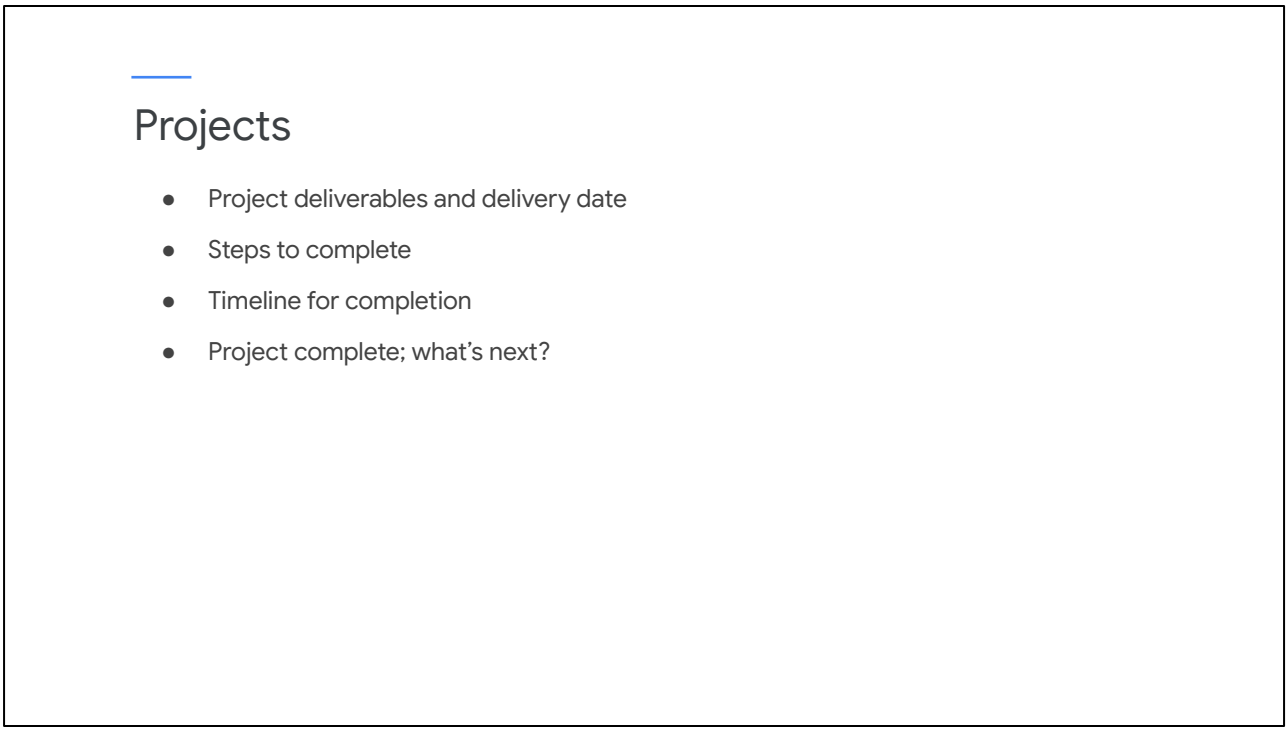
Apigee API Monetization provides an easy and flexible way to monetize your APIs. It allows an API provider to charge for API usage or pay back developers who generate new business revenue.

Monetization helps you provide a simple way for your app developers to sign up, set up billing, choose rate plans, and process credit card payments using your developer portal. Monetization also provides reports on usage, billing, and transaction activity for your app developers.

Monetization leverages API products to provide specific bundles of functionality to your app developers.

Monetization is available with Apigee's enterprise offerings.

Further discussion of API monetization is out of scope for this API Engineer training.




One important concept to remember is that a product is not the same as a project. You are probably familiar with how projects typically work.

When we think about projects, we typically think of a project's deliverables and delivery date. Just from the names, we understand that projects are all about delivery.

We begin a project by creating an ordered list of steps we need to complete before the project is finished.

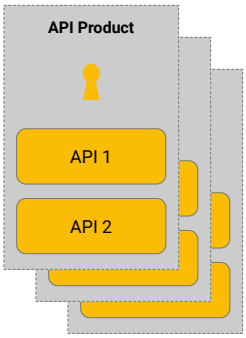
We then assign people to different tasks for the project and try to lay out a timeline.

Sometimes the delivery date is given to us even before we begin that process. Hopefully we are given enough resources to deliver the project within the requested timeline. Sometimes we choose to remove deliverables from the project in order to meet our deadline.

Delivering the project allows us to move on to the next project. We no longer focus on the completed project.

## Product ≠ project

- Products are designed based on business requirements.
- Products adapt to market needs over time.
- Products do not have a completion date.
- Recommendation: adopt a product mindset.




Products, on the other hand, focus on the outcome, not the delivery.

While a project is often represented by a scheduled list of activities that are required to solve a problem, products should be designed based on business requirements.

The person or team designing a product may have no idea how the product will be implemented, but that's ok

A product focus will be more likely to lead to a successful outcome, since we are focusing on the requirements of the API product rather than the process used to create it.

Products also need to adapt to market needs as they change. This means that we should be willing to modify our product over time. It can also mean that the new product you envisioned a few months ago might not be what you need now.

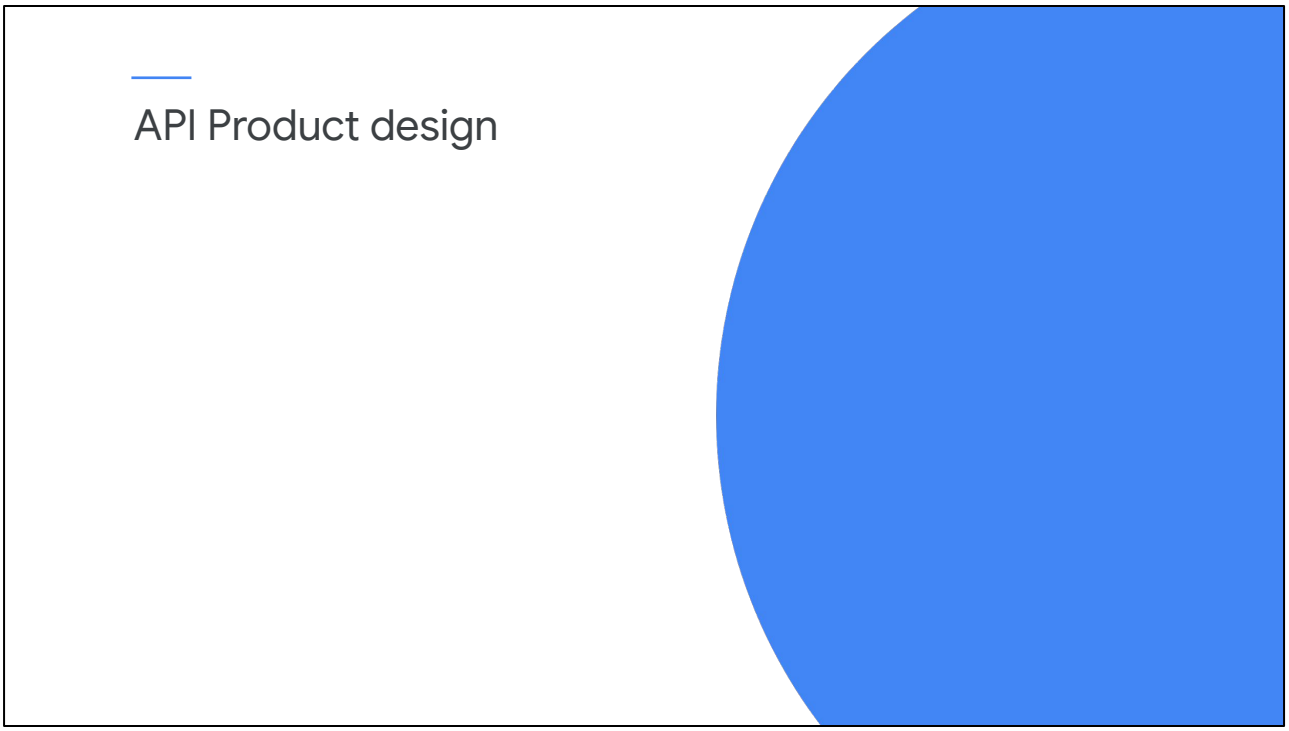
This is one of the problems we often see with IT projects when we design large projects up front for delivery months later. The team locks down requirements to avoid scope creep and give the team a chance to meet the delivery date, but the project often has reduced value when it is eventually delivered.

The need to have a product meet current business requirements means that products do not have a completion date. We're not saying that you never deliver a product. Rather, we don't think of products as something with an end date. We should keep

thinking about how market and business changes affect our ideal product, and make changes to our product as necessary.

It is rare to have a product that does not need some changes over time, so it is best not to think of your products as ever being truly complete.

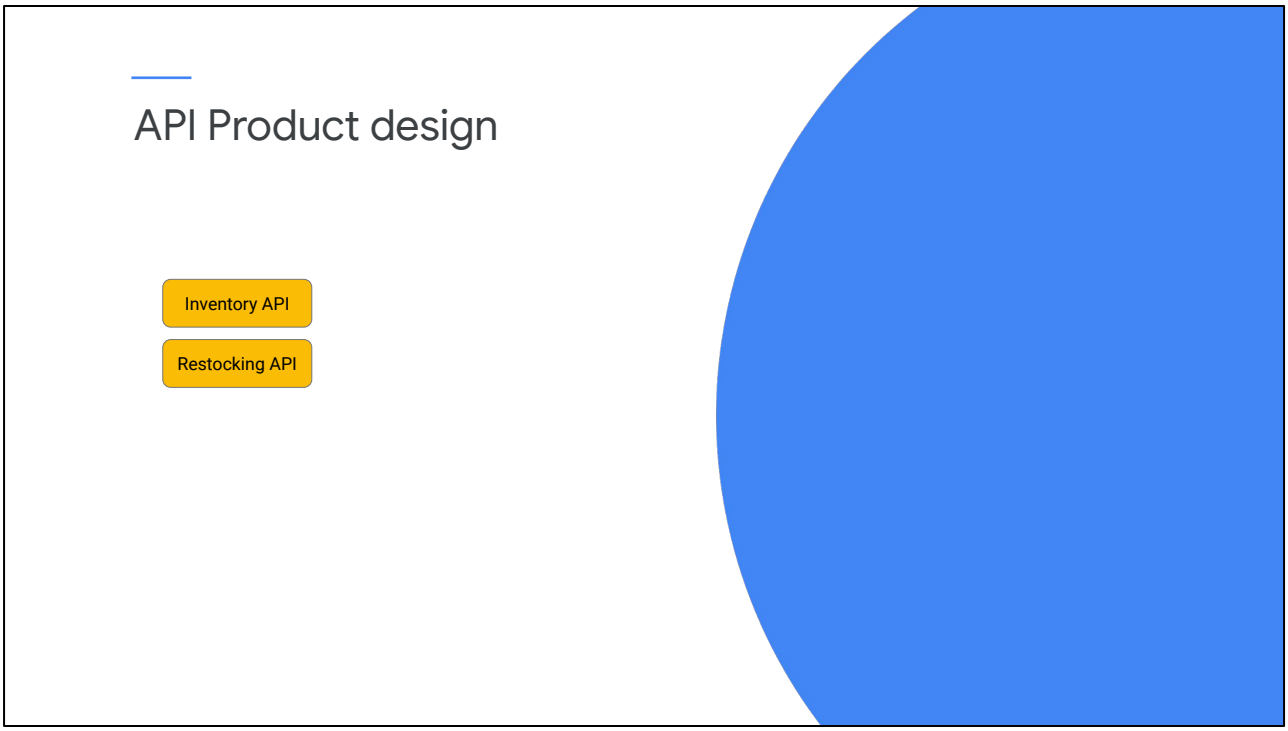
For these reasons, we strongly recommend that you adopt a product mindset for your APIs and API products, instead of thinking of them as projects. Our experience tells us that a product mindset leads to better adoption and customer satisfaction for your APIs and a more successful API program.




Let's discuss a scenario where multiple API products might be created for a company over time.

Imagine a company that sells household items at its stores around the world.

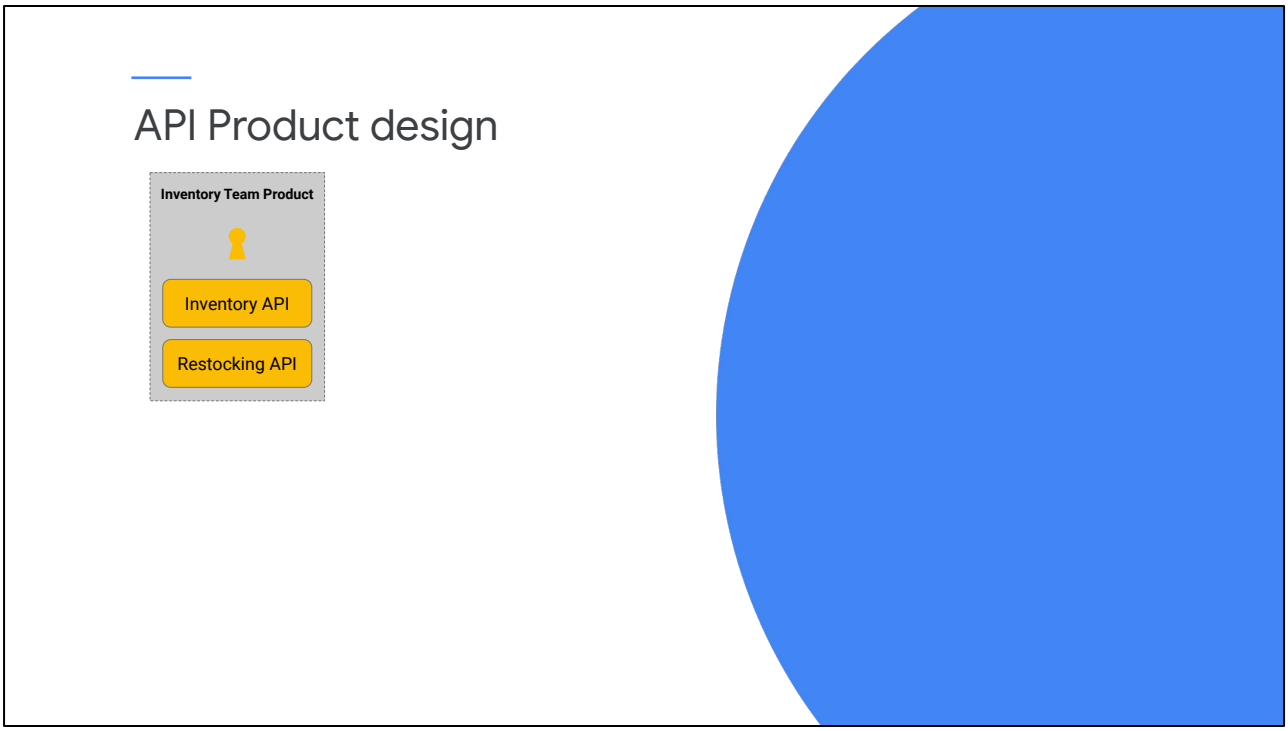
The inventory team is responsible for creating inventory tracking and restocking apps for running the business. The inventory team has APIs it uses to manage its work.




The inventory API can view and update inventory in the stores.

The restocking API can view, place, and track restocking orders for the stores.

The inventory team is currently the only consumer of the APIs, so the API product manager responsible for inventory...

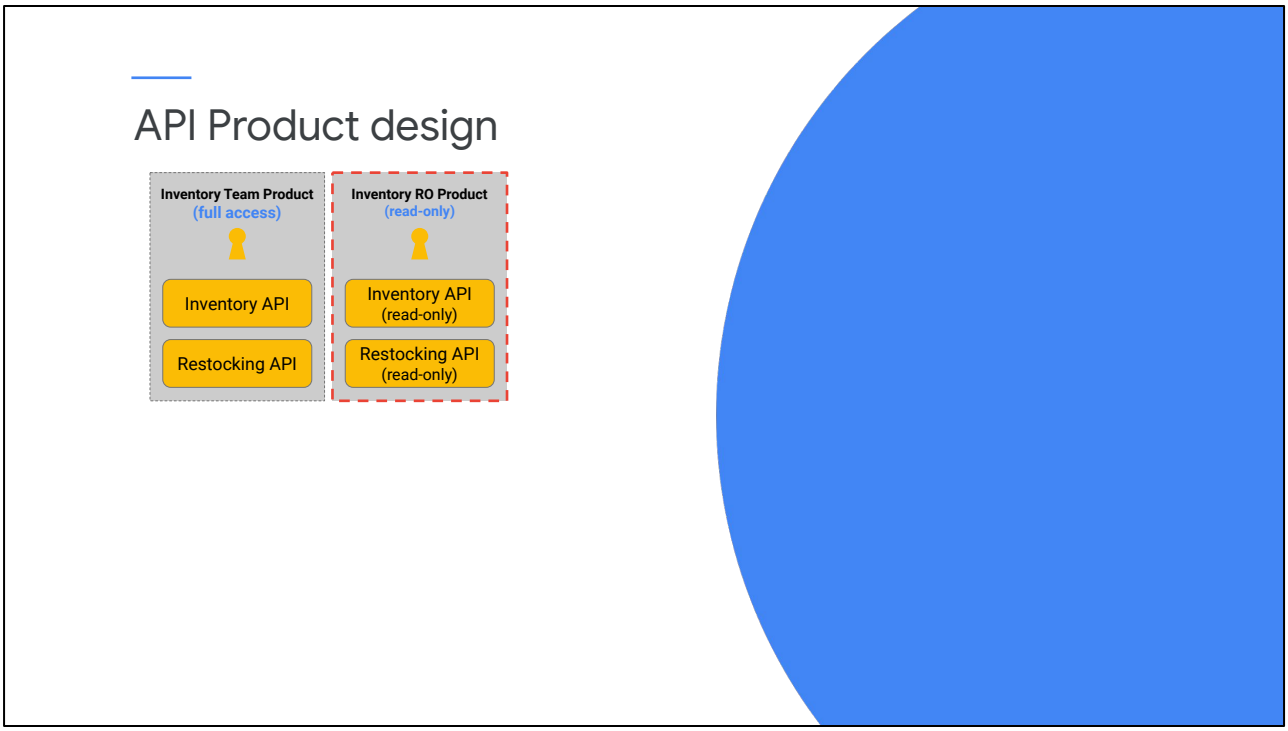



...creates a single API product that the inventory team can use for all its apps.

Soon, other internal teams see the valuable data that can be available using the Inventory's teams APIs and want access to the APIs to view the current state of inventory and restocking orders in stores.

The API team would love to have other teams use their APIs, but worry about giving other teams the ability to adjust inventory or create restocking orders.

The API product manager agrees that full read-only access for the rest of the company makes sense.

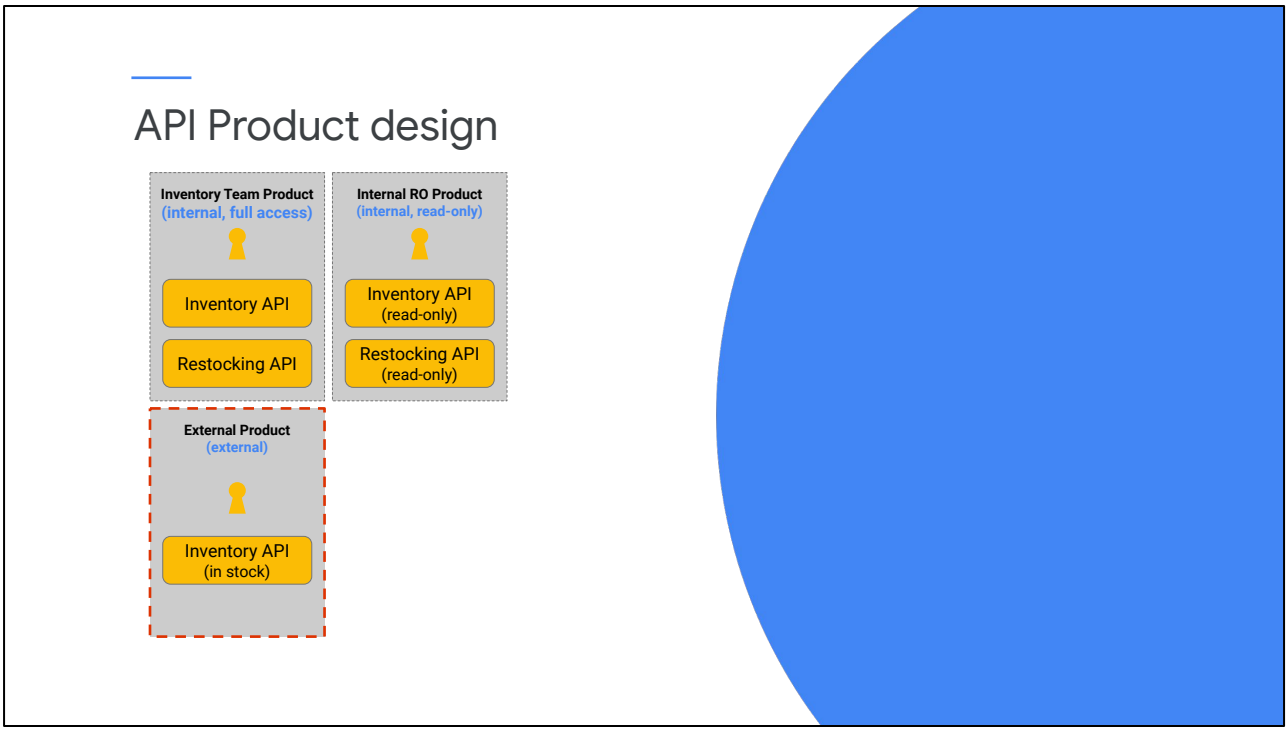



A second, read-only inventory product is created for internal use.

The two products are given a custom attribute indicating whether modifications are allowed. Now the API team makes a simple change to the API proxy to conditionally block attempts to modify state if the custom attribute is set to read-only.

Internal team apps given access to the new product can now check the current inventory and restocking orders, but attempts to update state, like creating a restocking request, are rejected before the request is sent to the backend.

The company is growing in popularity, and now external app developers want to provide their users in-stock information for popular products. The API product manager decides that providing this access will have a positive effect on sales and decides to give external developers access to information about whether a product is in stock at a particular store, but not the stock levels.



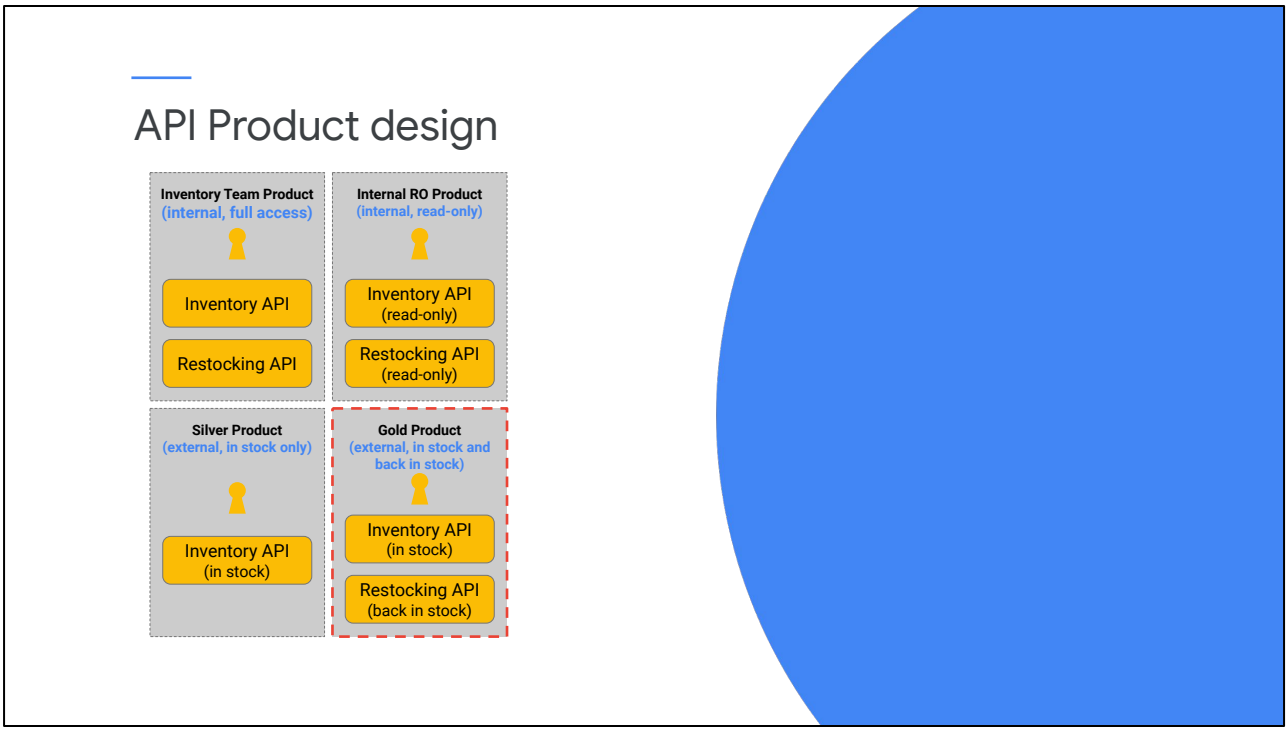

A third product is created to provide this external access. In this case, we want to make sure that external apps are never granted access to anything sensitive.

A second custom attribute is created, which indicates whether the product is for internal or external access, and the other two products are marked as internal.

The API team again modifies the API proxy: this external custom attribute is now checked up front, and if the product is external, any operation that is not approved for external use is blocked. This way, when new features are added to the proxy, they will not be available to external users unless access is explicitly granted.

The API team also realizes that there isn't an in-stock API call on the backend. The new call is created in the API proxy by calling the backend inventory API and returning that the item is in stock if the inventory level is greater than zero.

This new API product is successful, but some key partners want to be able to provide their users with expected dates that products will be back in stock.




Instead of giving all apps this ability, the API product manager decides to create a new API product that provides this access.

At this point, the external API products are also rebranded as the Silver and Gold products. The API team again provides some simple customization in the API proxy to allow only Gold product apps to access the restocking API.

You can imagine that other teams will want to productize their APIs as well.

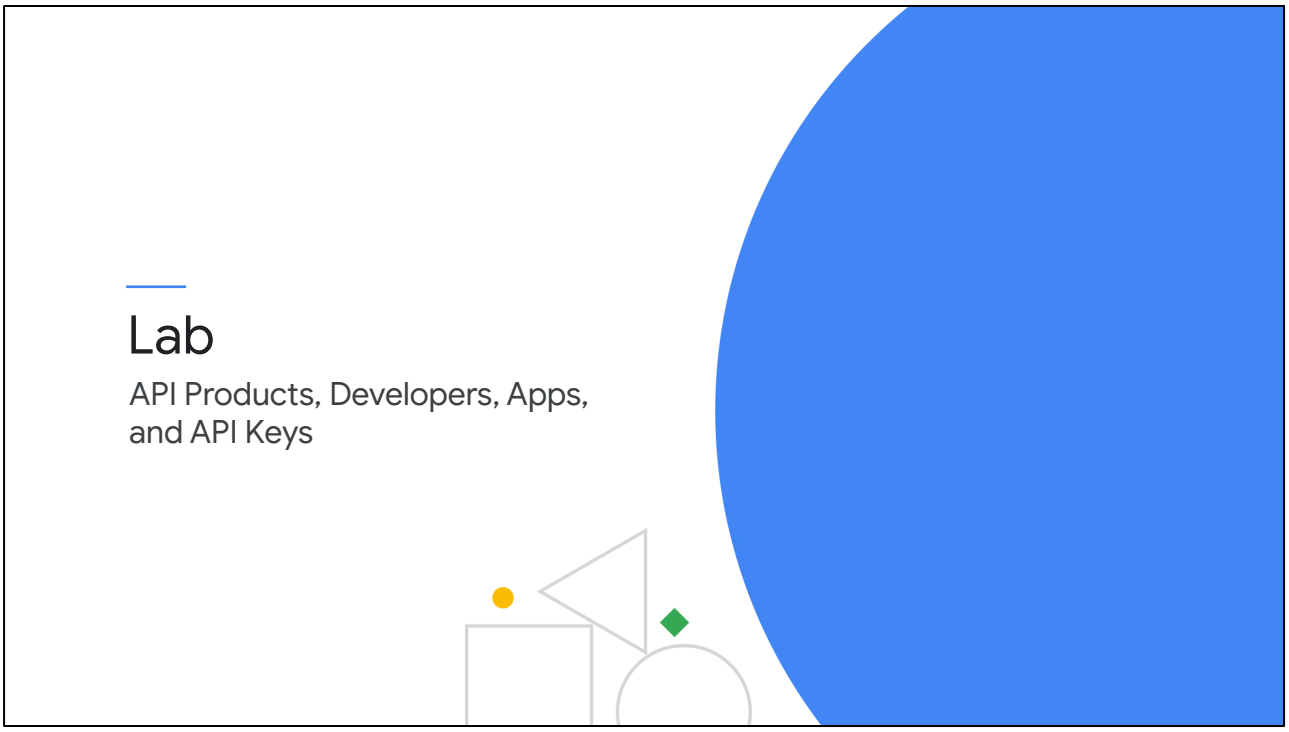
Remember that our product mindset focuses on the needs of the app developer, not the API development teams. As other teams add their APIs, we should focus on how app developers will want to use them.

APIs from many different teams may be combined into a single API product. App developers don't care which development team or business unit created the API, so your API product design should generally not be based on the implementation team.

We've seen that Apigee provides controlled access to APIs through the use of API products.

Note that all of these API products have been rolled out, including changes to levels of access and even new API calls, without making ANY changes to your backend APIs.

This is the power of Apigee. Apigee allows your API team to be agile and provide new features, even when your backend service teams can't roll out features quickly enough.




In this lab, you add a VerifyAPIKey policy to your retail proxy, requiring client apps to supply a valid API key. You will create an API product, an app developer, and an app. You will associate the API product with your proxy and associate the app with the API product. Then you test your API to confirm that API requests are only allowed when they supply a valid API key that is associated with your API product.


In this lab you explore the use of API product custom attributes to control access within your proxy. You will create a new proxy that requires an API key, create an API product with a custom attribute, and then control access within the proxy based on the custom attribute.


This lecture is the second of three lectures on REST API Design.

The first lecture introduced REST APIs and explained how to design RESTful APIs.

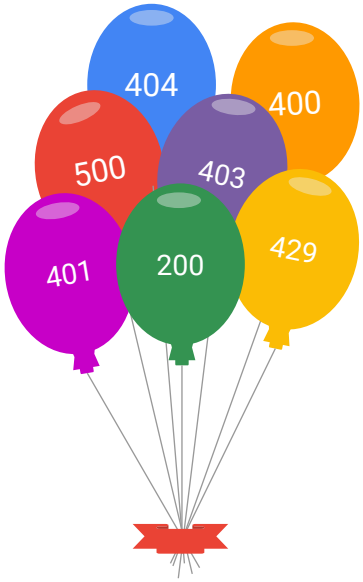
This lecture will teach you about API responses from REST APIs.

We'll learn how HTTP status codes should be used in your REST APIs and how we return responses that are useful for the app developer but don't expose too much information.

We'll also explore how we can use pagination for returning results across multiple calls to our APIs.

## Status codes

- Primary signals for success or failure.
- Have specific meanings.
- Should be consistent across your APIs.



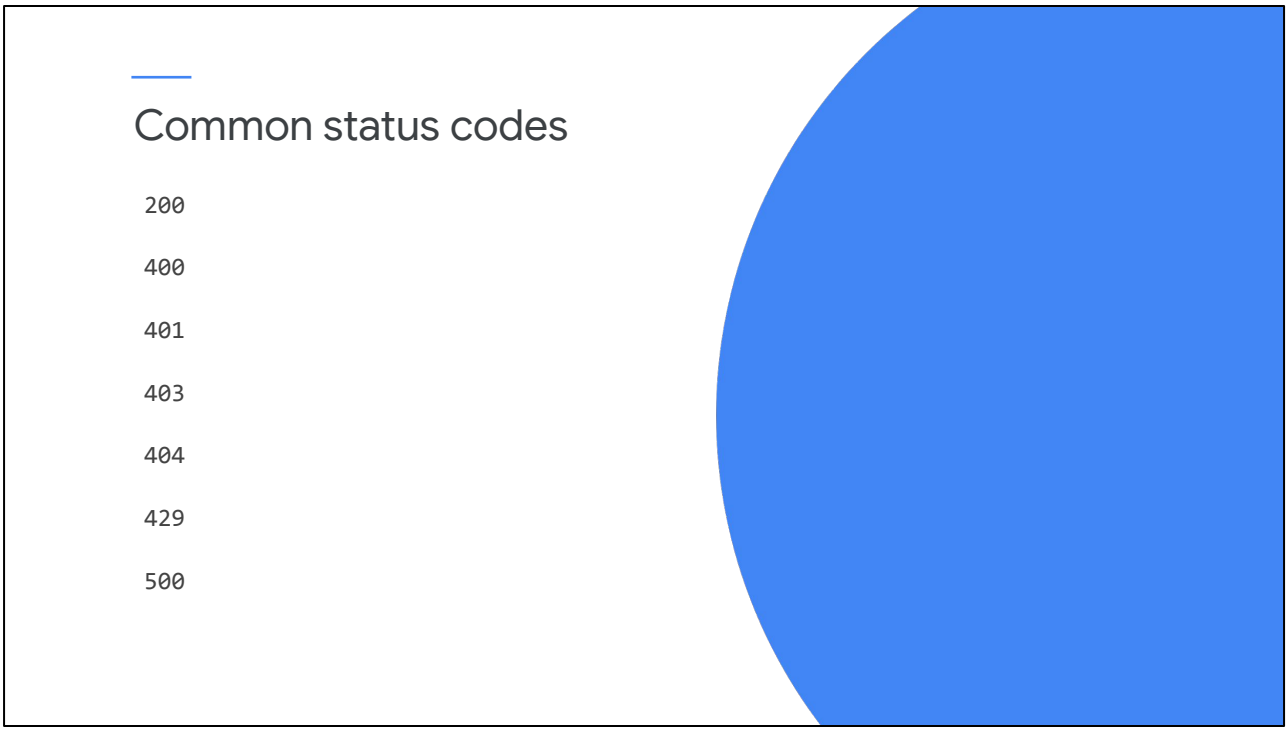

If you use web browsers, you probably have at least some idea what HTTP status codes are. You may also recognize some of the status codes shown here.

In web requests, as well as REST APIs, status codes should be used as the primary signal indicating whether an operation succeeded or failed.

Some other types of APIs, like SOAP, return operation success or failure in the payload, but this is an anti-pattern for REST APIs.

Status codes have specific, defined meanings. Part of the reason REST has been so successful is that it leverages common patterns and conventions of HTTP. You should adhere to the defined meanings when returning status codes for your APIs.

It is also important to be consistent with how you use status codes across all of your APIs. An app developer will find the second and third API easier to learn if those APIs follow the patterns of the first API.



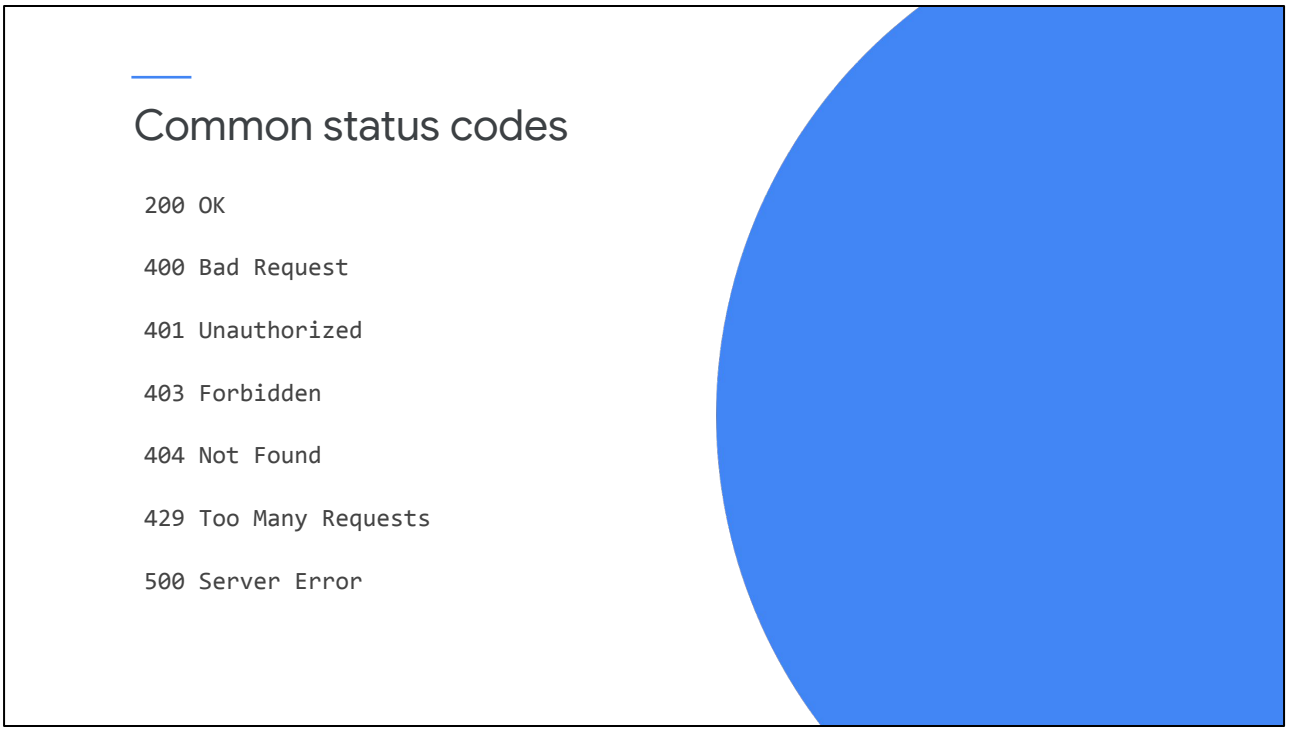

Listed here are seven status codes that are very common for REST APIs. If you have some experience with web development, you may have a good idea what these status codes mean.

Even if you are not a web developer, you'll probably recognize some of these.

An HTTP status code has an associated reason phrase that indicates what the status code means.

If you want to test your knowledge, pause the lecture now and try to guess the meaning of these status codes.

Let's start.




200 is OK. This means that the request was successful. The resource was found, and the requested action was taken.

The rest of the status codes on this page indicate some form of failure.

400 is Bad Request. The request made by the client didn't satisfy one or more requirements. However, the app developer should be able to repair the request and resubmit.

401 is Unauthorized. This actually means that the call requires authentication and the user has not successfully authenticated. This reason phrase is slightly confusing, since this error indicates a lack of authentication, not authorization.

403 is Forbidden. In this case the user has successfully authenticated, but the user is not allowed to perform the request for the specified resource or resources. If you are unsure what authentication and authorization mean, don't worry: we'll be discussing them in more detail in a later lecture.

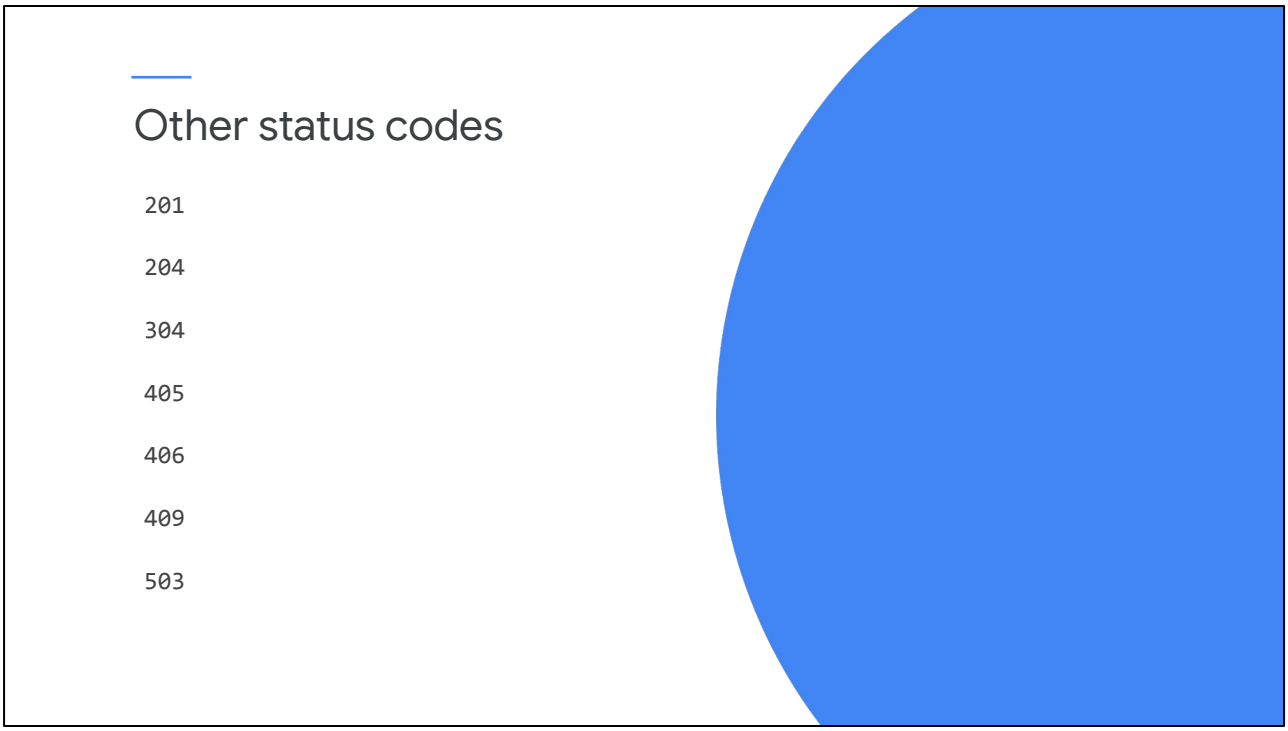
404 is Not Found, and everyone who uses a web browser has probably seen this error. For REST APIs, 404 means that the resource specified in the URL does not exist or at least is not accessible to the current user.

429 is Too Many Requests. This error is returned when the request is being rejected because the user, or all users, have made too many requests recently. We will see

that this error is used by Apigee for the spike arrest and quota policies. More on those policies later.

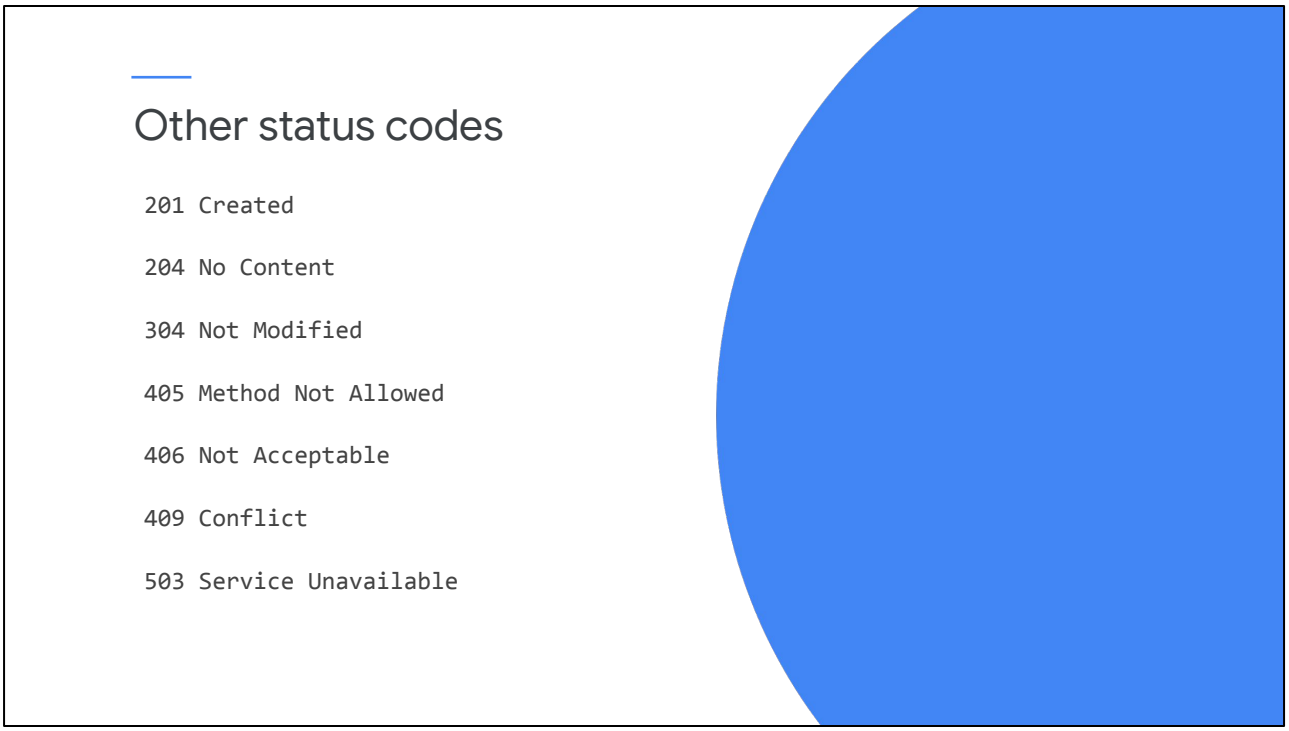
500 is Server Error. This should be used for an unexpected error on the server side. The expectation is that the caller might try the request again at a later time.

Many, if not most of the REST APIs you use or create should return these status codes.




Here are some status codes that you should also consider using for your APIs.

As before, pause the video now if you want to test your knowledge.




201 is Created. Like 200 OK, this status code indicates success. It is used when a resource is successfully created, generally using a POST.

Most APIs use 200 OK for successful creation of a resource. Whether you choose to use 200 or 201, be consistent across your APIs.

204 is No Content. 204 is a success code that indicates that there is no content to return in the response payload body. The request was successful, but there is nothing for the API to return. When returning a 204, the response cannot contain a message body.

304 is Not Modified. 304 is a success code that indicates that the version of the resource the caller already has is the most up-to-date version, so no payload response is necessary. This can be returned when you use cache headers. We will discuss this further in a later lecture.

405 is Method Not Allowed. This should be returned when the resource being requested is valid, but the requested verb isn't allowed for the resource.

406 is Not Acceptable. This should be returned when an incoming Accept header is asking for a response format that is not supported by the API.

409 is Conflict. This indicates that the request cannot be completed because there is a conflict. For example, when you read a resource and then attempt to update it, 409

could be returned if the resource was changed after the original read.

503 is Service Unavailable. This may be used when a service is temporarily unable to handle a request. For example, the service is down for maintenance. This should be used when the error is known to be temporary and service will be restored.

You may decide not to use these status codes, and there may be other status codes that you decide to use for your APIs. Just make sure that you are not using status codes in a way that conflicts with their established HTTP meanings.

## Status code ranges

- HTTP status code ranges have specific meanings:
  - 1XX - Informational
  - 2XX - Success
  - 3XX - Redirection
  - 4XX - Client Error
  - 5XX - Server Error
- Important to note the difference between 4XX client and 5XX server errors:
  - 4XX is the API saying "the error is due to your request."
  - 5XX is the API saying "the error is not your fault; you can't do anything about it."
- Don't automatically pass backend status codes through to the client.

You may have noticed that the status codes had values like 200, 400, and 500. Status code ranges have defined meanings.

Status codes in the 100s are informational -- neither success nor failure. Processing is not complete.

Status codes in the 200s indicate success. The request was received, understood, and accepted.

Status codes in the 300s indicate redirection. The client must take further action to fulfill the request. In the case of 304, the client would redirect to, or use, the cached value.

Status codes in the 400s are client errors. These indicate that the client is the source of the problem. If the client fixes the request, it can be successful.

Status codes in the 500s are server errors. Through no fault of the client, the server cannot service the request.

Note the 400 and 500 ranges. 400 says that the error is the fault of the client, and 500 says that the error is the fault of the service.

It is important to use status codes in the correct ranges. Apps will sometimes choose how to notify the user based on the value of the status code.

This also means that backend status codes are sometimes not appropriate to return to the client. Say the backend service is returning a status code of 400, Bad Request. It may be the API proxy that is not formatting the request correctly. And this might be happening even though the app sent in the correct format as expected by the proxy. So returning a 400 to the app would indicate that the app could fix the problem, which is not the case.

Make sure you always think about the meaning of the status code being returned to the app developer when you are designing your APIs.

## Bad response #1

```
GET /users/123
400 Bad Request
{
  "error": "Bad Request"
}
```

Let's look at some bad responses.

Look at this response. Try to figure out what is wrong with the API response and how it could be improved.

Pause the video if you'd like to think about it a bit before we reveal the answer.

## Bad response #1

- No context for how to repair the request.
- Better solution:

```
GET /users/123
400 Bad Request
{
  "error": "Bad Request"
}
```

```
400 Bad Request
{
  "error": "Missing query parameter: q",
  "message": "See https://apigee.org/api/getUser"
}
```

The 400 status code for this response indicates that the error is due to the requestor, but the response gives no information indicating how to fix the problem.

If you were the app developer, imagine trying to fix the issue without having any information about what was wrong with the request.

The improved solution tells the app developer that a required query parameter is missing and gives the developer a link with information about how to make the request.

## Bad response #2

```
GET /users/123
404 Not Found

GET /users/456
403 Forbidden
```

Look at these two requests. We are only showing the status codes, not the responses.

What's the issue here?

## Bad response #2

```
GET /users/123
404 Not Found

GET /users/456
403 Forbidden
```

- Inadvertent leakage
  - “Now I know user 456 exists, so I’ll start trying to hack that user.”
- Better solution is to always return

  `404 Not Found`

These responses leak information that could be useful for someone trying to compromise the API. There is no reason to leak the information that user 456 exists if the caller does not have the authorization to access the resource.

It is better to return 404 if the user cannot access the resource, whether the resource exists or not.

## Bad response #3

```
GET /users/123
500 Server Error
{
 "error": "DB error",
 "message": "Tibco failure querying
MySQL database at 10.3.4.33"
}
```

OK, one more.

This one should be easy.

## Bad response #3

```
GET /users/123
500 Server Error
{
 "error": "DB error",
 "message": "Tibco failure querying
MySQL database at 10.3.4.33"
}
```

- Don’t leak internal server details.
- Better solution:

```
500 Service Unavailable
{
 "error": "User service unavailable, please try
again later",
 "message": "Report error by emailing
errors@apigee.org",
 "correlationId": "C3358-23"
}
```

This response is leaking information about what kind of systems are being used in the backend, as well as an internal IP address. You don't want to help hackers know what is running in your backend: they could try to exploit known vulnerabilities with this information.

The better solution returns information to the user with a correlation id which is used when logging information in the backend service. This would allow your engineers to debug a reported issue without unnecessarily exposing sensitive information to the caller.

Note that this type of error response may be returned through your API if you don't take care to block it at the API proxy layer. The backend service may have been originally designed to be used only on the company's internal network. This type of error would be useful for quickly resolving issues with the API.

Do not return backend error messages from your API proxy unless you are sure they are returning safe information.


Pagination is a way to return a subset of data at a time. We call the subset a *page*. These results are ordered, so the next set of results, according to the order, should be returned with the next page.

When an API call is a search or query that can return multiple resources, we generally expect to receive paginated results. If we've chosen our ordering correctly, the most useful information should be in the earliest pages retrieved.

If there are potentially many matching resources, pagination can make sure we don't return responses that are too large.

Pagination can be very useful if the resources being returned are unusually large, or network bandwidth is important, as is the case for mobile apps.

## Example pagination response

```
GET /users?offset=21&limit=10

{
  "resources": [
    {"id":"134C3", "firstName":"Bob", "lastName":"Roberts", // resource fields for display
     "href": "https://.../users/134C3" },                   // link to get full resource
    ...
  ],
  "pagination": {
    "offset": 21,                                           // index of first resource
    "count": 10,                                            // number of resources returned
    "totalResults": 245,                                    // total number of resources
  },
}
```

Here is an example of how a pagination response might be implemented.

Note the query parameters being passed into the request. Offset indicates the first resource to retrieve, the twenty-first in this case. Limit is 10, indicating that the next 10 resources should be returned.

You could imagine that a search or filter could also be used to select a subset of the resources, and then the limit and offset would be used to iterate through that subset.

## Example pagination response

```
GET /users?offset=21&limit=10

{
  "resources": [
    {"id":"134C3", "firstName":"Bob", "lastName":"Roberts", // resource fields for display
     "href": "https://.../users/134C3" },                   // link to get full resource
    ...
  ],
  "pagination": {
    "offset": 21,                                           // index of first resource
    "count": 10,                                            // number of resources returned
    "totalResults": 245,                                    // total number of resources
  },
}
```

The list of entities or resources is generally implemented as an array of objects.

## Example pagination response

```
GET /users?offset=21&limit=10

{
  "resources": [
    {"id":"134C3", "firstName":"Bob", "lastName":"Roberts", // resource fields for display
     "href": "https://.../users/134C3" },                   // link to get full resource
    ...
  ],
  "pagination": {
    "offset": 21,                                           // index of first resource
    "count": 10,                                            // number of resources returned
    "totalResults": 245,                                    // total number of resources
  },
}
```

Each object contains the fields for the resource that would be needed for presenting a list of resources, with the ability to select a resource and retrieve more details for that resource.

In this case, you see the ID, first name, and last name for the resources. Other use cases might need additional fields.

In addition, the objects in this response contain a link, or reference, that could be used with GET to retrieve the details for the specific resource.

## Example pagination response

```
GET /users?offset=21&limit=10

{
  "resources": [
    {"id":"134C3", "firstName":"Bob", "lastName":"Roberts", // resource fields for display
     "href": "https://.../users/134C3" },                   // link to get full resource
    ...
  ],
  "pagination": {
    "offset": 21,                                           // index of first resource
    "count": 10,                                            // number of resources returned
    "totalResults": 245,                                    // total number of resources
  },
}
```

This response also has a pagination section that returns information about the resources returned and the total number of resources in the results.

## Pagination types

- Offset and limit
  - Works well for most UI use cases, but may not allow for consistent paging through all resources.
  - Updating resources can cause their location in the list to change, resulting in resources being returned multiple times or never at all.
- Value-based paging
  - A “last” parameter is returned with the last resource retrieved.
  - Specifying the parameter retrieves resources starting with the next one.

The example response we just saw was using offset and limit to page through the results. This method works very well for UI-based use cases, where you are letting the user of an app page through the results.

Based on the type of sorting used in your results, though, you might not consistently page through all the results, as updates to the resources could cause them to be seen again on a later page, or be skipped altogether if the changes to the resource moved it to a previously seen page.

For use cases where we want to allow the caller to retrieve all of the results, we might implement value-based paging. Value-based paging returns a field that can be returned with the next call to retrieve the next group of resources.

An example is using a last parameter, which might return the creation timestamp or id of the last resource retrieved. Passing this last parameter in to the next request would retrieve the next set of resources that follow the last parameter from the previous request. This can result in the caller predictably seeing every resource in the list.

If the use case requires traversing through all of the resources, you might choose to use value-based paging.


You have learned about API products, app developers, apps, and API keys.

We discussed API product strategies and REST API responses and HTTP status codes.

You also completed a lab that added API key protection to your retail API, and a lab that used API product custom attributes to control an API proxy.


Thank you for taking the API Design and Fundamentals course.

I hope you have become more comfortable and knowledgeable about creating and designing APIs using Apigee.

During this course you learned about Apigee and the API lifecycle.

We discussed REST API design, API first design, and OpenAPI specifications.

You learned about API proxies and policies.

Finally, you learned about API products, which are used to package APIs for use by app developers.

You did several labs in which you created and tested API proxies.

We recommend you continue with the next course in this series, API Security on Google Cloud's Apigee API Platform.

In the API Security course, you'll learn about API security concerns, and how Apigee will help secure against them.

You'll learn about OAuth, an authorization framework for APIs.

You'll also learn how to protect your APIs against content-based and internal security threats, and you will complete labs to add security to your retail API proxy.

Please join us in the next course!