**Purpose**

We don't just build an API because it is a fun thing to do. We build APIs to achieve a purpose. They help us or our users do something that we want to do more efficiently. This might seem obvious, but let's be honest: we forget to do this far too often.

Knowing that you need to do something is an important first step. Unfortunately, many talks, articles and books stop there. But what good does that do you? You are convinced that it makes sense to design an API with the purpose in mind, but what is the purpose? What do you do if you don't know? How do you figure out the purpose of your API?

**Figuring out the purpose**

You want to figure out the purpose of your API, but how do you do that? There are a few simple strategies that you can use for this. The first thing is to ask questions. If you or your team is writing an API, you were probably asked to do it by someone. Perhaps your manager or the product manager or one of the clients. Someone asked you to create the API. Talk to that person! Why do they want the API? What do they want to do with it? What problems do they think it will solve? Ask them these questions and see if you can figure out some of what the purpose of the API is.

**Personas**

Another simple strategy that you can use to think of personas. A persona is simply a made-up person that you use to help you think through who might be using your API. For example, you might have a persona representing a user in your company who will be using the API to develop user interfaces, or you might have a persona of a software engineer who works for one of your clients and is using the API to develop a custom application. Thinking through different kinds of users and considering how they might interact with your API will help you with understanding the purpose that it serves. When creating a persona, think about things that go beyond just the technology that the person might use or understand. Think about things like the goals and motivations they might have and the things that might frustrate them. It's also helpful to think of some personal details that might make them a bit more relatable to you. For example, you could think about whether they are a dog person, or whether they have kids. Is there some other thing that makes them unique? In other words, what kind of person are they? Writing down details like this can seem a bit silly at times, but it helps you to better empathize with this persona and as you are able to do that you will be more able to put yourself in their shoes and understand the kinds of things that are important to them. The more you understand this, the better you will be able to understand the purpose of your API.

**The why**

At the heart of figuring out the purpose is the question why. Why are we making this API? The why is almost always about solving a problem. A great way to figure out the purpose of an API is to figure out what problem it solves. Does it make it easier to write UI elements? Does it enable clients to customize the application? Does it enable third-party developers to use your platform? Does it simplify integrations with other applications? What problem is your API solving? Answer these questions and you will be well on your way to knowing the purpose of your API.

**Important note**

This exercise of figuring out the purpose of an API doesn't just apply to new APIs. If you are working with an existing API, there is a lot of value in understanding the purpose. It may be too late to radically alter the design of the API, but there is no more important threat to quality than not actually helping people solve the problems they need to solve. If nothing else, understanding the purpose of an existing API that you are testing will help you figure out which bugs are important and which ones might not matter as much. It takes some skill to figure out which bugs are urgent, and which are not, and understanding the purpose of the API helps with that.

1. Identify at least two key stakeholders for the API. Do this by asking the question "who wants (or wanted) this API built?" Write down these stakeholder names.
2. If possible, talk to those stakeholders and ask them what they think the purpose of this API should be and why they want to build it. Write down their answers.
3. Create preferably two (but at least one) personas that list out the kinds of people that you think will be using the API. What skill level do they have? What work are they trying to accomplish? How will your API help them?
4. Write down what problem(s) you think the API will solve.
5. Now, take all the information that you have gathered and look through it. Distill it down into two or three sentences that explain the purpose of the API.

**Usability of an API**

Usability is about the balance between exposing too many controls and too few. This is a very tricky thing to get right. On the extremes, it is obvious when things are out of balance. For example, the Metropolitan Museum of Art has an API that gives you information about various art objects in their possession. If all the API did was provide one call that gave you back all that data, it would be providing too few controls. You would need to do so much work after getting the information that you might as well not use the API at all. However, if on the other hand, the API gave you a separate endpoint for every piece of meta data in the system, you would have trouble finding the endpoint that gave you the particular information you wanted. You would need to comprehend too much in order to use the system. You need to think carefully about this if you want to get the balance right. Make sure your API is providing users with specific enough data for the things they need (this is where knowing the purpose comes in handy) without overwhelming them. In other words, keep it as simple as possible.

**API Structure**

One thing that can help create a usable API is to use only **nouns** as endpoints. If you want users to be able to understand your API, structure it according to the objects in your system. For example, if you want to let API users get information about the students in a learning system, don't create an endpoint called /getAllStudents. Create one called /students and call it with the GET method. Creating endpoints based on nouns will help you better structure your data. For example, if you have /students as an endpoint, you can easily add an endpoint for each student at /students/studentId. This kind of categorization structure is another helpful API design principle to keep in mind. Creating a structure like this maps the layout of the API onto the kinds of things that the API user needs information about. This makes it much easier to know where to find the relevant information.

A structure like this works nicely, but does it really match up with how users will interact with the API? If I am looking for information about a student, am I going to know what their ID is in the API? Perhaps, but more likely I will know something like their name. So, should we modify the structure to have an additional endpoint like /students/name? But what if we are looking at all the students of a certain age? Should we add another endpoint /students/age? You can see where I am going with this. It can get messy pretty quickly.

This is where **query parameters** are helpful. A query parameter is a way of getting some subset of the category based on a property that it has. So, in the examples that I gave earlier, instead of making "name" and "age" be endpoints under the "students" category, we could just create query parameters. We would call /students?name='JimJones' or /students?age=25. Query parameters help keep the endpoints simple and logical but still give the users the flexibility to get the information they are interested in in an effective way

**RESTful API Modeling Language (RAML)**

**RAML**, which stands for RESTful API Modeling Language, is an API specification language that, as the name implies, helps with modeling APIs.

Getting started with RAML is as easy as opening a text editor and typing in some text. RAML is meant to be human-readable and so the specification is written in a simple text-based format. RAML is also structured hierarchically, which makes it easy to create the kind of usable API structures.

API design does not only apply to new APIs that you create. In fact, using the principles of API design when testing an existing API is a great way to find potential threats to the value of that API.

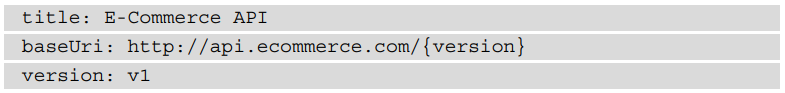
**Case study – Designing an e-commerce API**

Let's imagine that we want to design an API for a very simple e-commerce application. This application has a few products that you can look at. It also allows users to create a profile that they can use when adding item to their cart and purchasing them. The purpose of this API is to expose the data in a way that can be used by both the web and mobile application user interfaces. Your team has just been given this information and you need to come up with an API that will do this.

The first thing we need is to create a file and tell it what version of RAML we are using. I did this by creating a text file in Visual Studio Code (you can use whatever text editor you prefer) called E-Commerce\_ API-Design.raml. I then added a reference to the top of the file to let it know that I want to use the 1.0 version of the RAML specification:



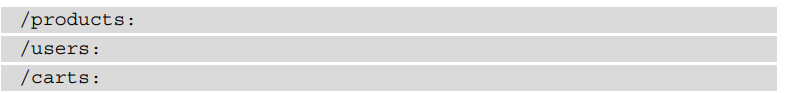
I also need to give the API a title and to set up the base URI for this API, so next I defined those in the file:



**Defining the endpoints**

This is a made-up API so that baseURI reference does not point to a real website. Notice also how the version has been specified. Now that I have defined the root or base of the API, I can start to design the actual structure and commands that this API will have. I need to start with the purpose of this API, which is to enable both a website and a mobile app. For this case study, I am not going to dive too deep into things like creating personas. However, we do know that this API will be used by the frontend developers to enable what they can show to the users. With a bit of thought, we can assume that they will need to be able to get some kind of product information that they can show the users. They will also need to be able to access a user's account data and allow users to create or modify that data. Finally, they will need to be able to add and remove items from the cart.

With that information in hand about the purpose of the API, I can start to think about the usability and structure of this API. The frontend developers will probably need a **/products** endpoint to enable the display of product information. They will also need a **/users** endpoint for reading and maintaining the user data and a **/carts** endpoint that will allow the developers to add and remove items from a cart.



*These endpoints are not the only way that you could lay out this API***.** For example, you could fold the carts endpoint into the users one. Each cart needs to belong to a user, so you could choose to have the cart be a property of the user if you wanted. It is exactly because there are different ways to lay out an API, that we need to consider things like the purpose of the API. In this case, we know that the workflow will require adding and removing items from a cart regularly. Developers will be thinking about what they need to do in those terms and so to make them call a "users" endpoint to modify a cart would cause extra data to be returned that they do not need in that context, and could also cause some confusion

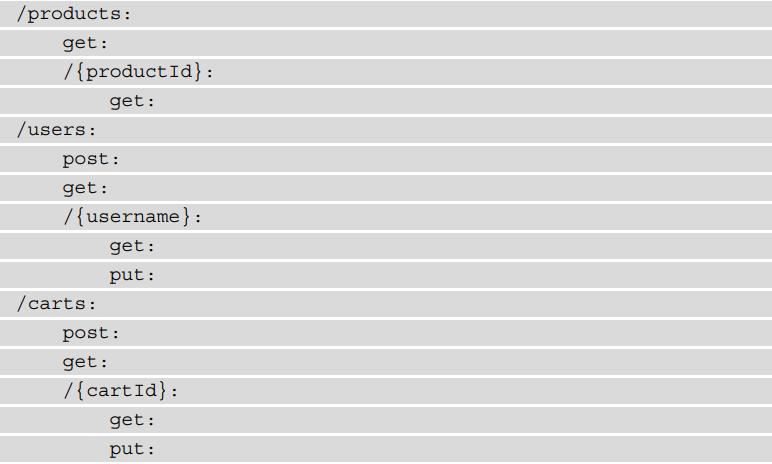
**Defining the actions**

Of course, we need to be able to do something with these endpoints. What actions do you think each of these endpoints should each have?

My initial thought was that we should have the following actions for each endpoint:



Think about this for a minute, though. If I only have one endpoint, /carts, for getting information about the carts, I need to get and update information about every cart in the system every time I want to do something with any cart in the system. I need to take a step back here and define this a little better. The endpoints are plural here and should be representing collections or lists of objects. I need some way to interact with individual objects in each of these categories:

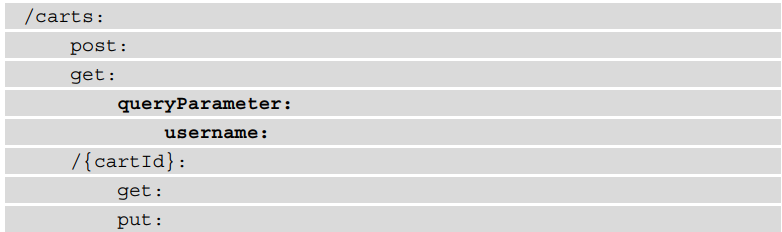


Here, I have defined **URI parameters** that enable users to get information about a particular product, user, or cart. You will notice that the **POST** commands stay with the collection endpoint as sending a POST action to a collection will add a new object to that collection. I am also allowing API users to **GET** a full list of each the collections as well if they want.

**Adding query parameters**

Looking at this from the perspective of the users of the API, I think it would be helpful to use a query parameter in the carts endpoint. When a user clicks on a product and wants to add that product to their cart, the developer will already know the product ID based on the item the user clicked on. However, the developer might not have information about the cart ID. In this case, they would have to do some sort of search through the carts collection to find the cart that belonged to the current user. I can make that easier for them by creating a query parameter. Once again, I am using the design principles of usability and purpose to help create a good model of how this API should work

In RAML, I just need to create a query parameter entry under the action that I want to have a query parameter:



**Summary:**

Using the API you have selected, work through the following steps to apply the principles of API design:

1. Add each of the endpoints to a RAML file. Make sure to follow the hierarchy of the API in that file.
2. Spend a bit of time thinking about what the purpose of the API is and reflecting on if the structure that you see here fits with that purpose. What other ways might you design the API? Could you improve the layout to better fit the purpose?
3. If you were designing this API, what actions and query parameters would you give to each endpoint? Create a copy of the file and fill it in with what you think you would do with this API.
4. In the original file, add in the actual actions and query parameters that the API has. How do they compare to the ones that you made in the copy of the file?