An Introduction to REST

By: Mosh Hamedani

Client-server Architecture

Many of the applications you use on a daily basis have a client side, which is the one you use, and a server side that you don't see! This is what we call *client/server architecture*.

All websites use this client/server architecture. So, when you head over to google.com, your browser (client) talks to google's servers. It can be argued that your machine is the client, and not your browser. But these arguments are like religious arguments. They make more harm than bring any values!

URLs

These servers have endpoints that are accessible by the clients. These endpoints are identified using what we call a *Unified Resource Identifier* (URI) or *Unified Resource Locator* (URL). Here are some examples:

http://google.com
http://mywebsite.com/api/customers/1

HTTP Requests and Responses

The communication between the client and the server happens using the HTTP protocol. It defines a basic language and a set of rules for the client and server to communicate. So, the client sends a message to the server, which we call an *HTTP Request*. And the server produces a result, which we call an *HTTP Response*.

An Introduction to REST

By: Mosh Hamedani

HTTP Verbs

We have different types of HTTP requests and their type is determined by what we call a *verb.* These verbs include:

GET: for getting a resource from the server. This resource can be an HTML file, an image, etc.

POST: for posting or creating a resource on the server. So, when you fill out the form too book your travel insurance, an HTTP POST request is sent to the server. The body of this request includes data that you filled out in the form.

PUT: for updating a resource on the server. The way I remember the difference between POST and PUT is the letter U in PUT. It always reminds me of update!

DELETE: obviously for deleting a resource.

There are a couple more verbs that are not commonly used.

JSON

So what is this JSON? It stands for *JavaScript Object Notation*. It's a simple syntax (or notation) that we use to represent an object in JavaScript. Here is an example:

```
{
    id: 1,
    name: “Mosh”
}
```

In JavaScript, we don't have types. So, we can put these key/value pairs (id and name) between curly braces and we have a JSON object!

Or we could put many objects in an array:

```
[
	{ id: 1, name: “Mosh” },
	{ id: 2, name: “Bob” }
]
```

JSON is a very popular format these days. Why? Because it’s very lightweight. It doesn’t have all these annoying opening and closing tags of XML:

```
<SomeTag>…</SomeTag>
```

RESTful Services

And what the hell is this RESTful? The academic definition is *Representational State Transfer*. Does it make sense to you? It doesn’t make sense to me! It was originally the thesis of a PhD student. That aside, in simple pragmatic terms, it is a convention for building standard data services.

Let’s say you have a website like Facebook. You want to expose data services that can be accessed at certain endpoints. Developers around the world can consume your data services and build new experiences for the users. RESTful services, also called RESTful APIs, follow a standard convention.

Quite often (not always) we expose data services at a URL that includes the word “api”:

```
http://github.com/api/…
```

What comes after API is what we call a *resource*, or more accurately a resource collection. Let's say you're one of the developers at GitHub. You want to expose the list of GitHub users to the world. You could expose them here:

```
http://github.com/api/users
```

RESTful Conventions

Getting a Collection of Resources

To get all the users we should send an HTTP GET request as follows:

```
GET   http://github.com/api/users
```

Getting a Single Resource

To get only one user, we append the ID of the user in the URL:

```
GET   http://github.com/api/users/1
```

Creating a Resource

To create a user, we send an HTTP POST request to the server. We add the data about the user as JSON in the body of the request:

```
POST   http://github.com/api/users
```

Note that we are posting the user to the "users" collection.

Updating a Resource

To update a user, we send an HTTP PUT request to the server. In the URL, we need to specify the ID of the user we're going to update. And in the body of the request, we add the updated user serialized as JSON.

```
PUT   http://github.com/api/users/1
```

Note the difference between the URL in POST and PUT. In POST, we post the user object to users collection. So the URL does not include ID of the user. But in PUT, we need to specify that.

Deleting a Resource

To delete a user, we send an HTTP DELETE request to the server. Similar to HTTP PUT, we need to specify the ID of the user in the URL:

```
DELETE   http://github.com/api/users/1'
```

But how…

But how are we going to send an HTTP request and get a response? Well… that depends on the kind of application you're building. In Xamarin, we use a library called Microsoft.Net.Http that you're going to learn in this lecture. It abstracts all the complexities of sending an HTTP request and getting a response. So, we have a class called **HttpClient** and this class has methods for sending various kinds of HTTP requests.

RESTful Convention Continued

What you've seen so far covers a small part of the RESTful convention: the request part. But what about the responses? There are a bunch of rules for the server to respond.

For example, if it cannot find the requested resource, it should set the status of response to 404. Or, if it successfully processes a request, it should set the status code to 200. There are lots of codes and you do not need to memorize any of them. No one does really, unless they got nothing better to do in life!

All you should understand now is what RESTful convention is all about!