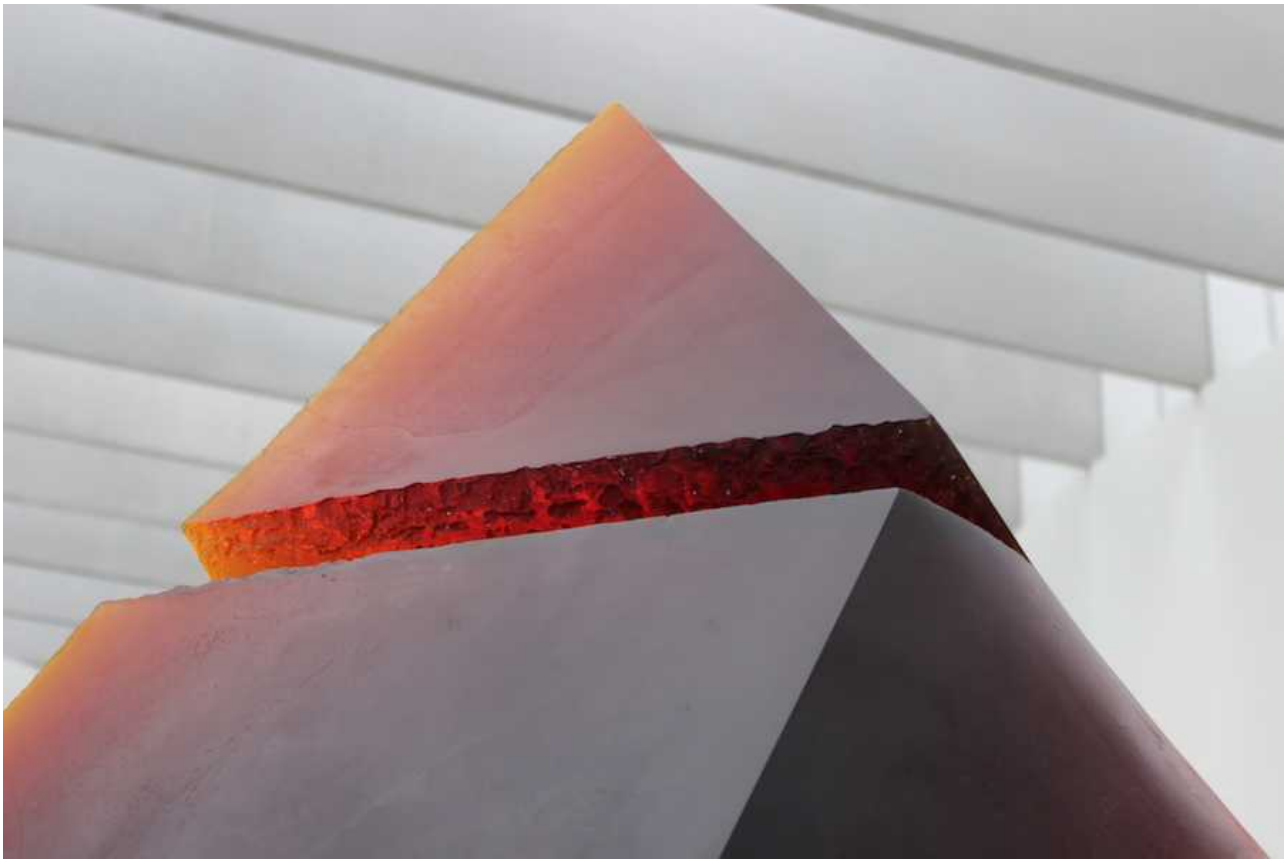
Nobody introduced me to the API

MAY 10, 2017 BY ROBIN WIERUCH - EDIT THIS POST



It might be a common issue in teaching computer science at universities: While you learn about bubble sorts, lambda calculus and permutations, nobody mentors you about common developer subjects. In the early days at university, I often wondered: *What is an API?*

Not everyone is fully aware of it, but APIs follow us through a multitude of programming contexts. They surround us. When I take you on the journey of how I got to know the API, you may get the idea what I mean by using APIs unconsciously most of the time. They give you an interface ranging from remote services, components in a web application to sensors in mobile devices. Interfaces are all around us.

However, there is a knowledge gap when it comes to APIs. It starts when teaching programming. Teachers assume that APIs are common knowledge. The abbreviation pops

up in tutorials, courses and books without any introduction. But it isn't common ground that teachers try to build on. Everyone is referring to this mysterious API, but nobody explains what it really is. My own articles about learning programming are not excluded from this flaw. That's why I want to take you on this journey of mine when I got to know APIs.

The article is my attempt to take you on a journey by giving you an introduction to APIs. Rather than giving you a technical Wikipedia reference, this journey attempts to wrap the subject of APIs into a story that I experienced myself when I studied computer science. The journey shows how a multitude of concrete fragments, like APIs in different programming contexts, can become something abstract, like interoperability between entities, in the end. If you are curious how to convey this information in JavaScript afterward, checkout the great blog post by Tania Rascia: How to Connect to an API with JavaScript.

Apart from the introduction to APIs, the journey might also give you insights into other topics: studying computer science, the general struggle and clumsiness when learning programming and the experience of perceiving the bigger picture eventually.

Hopefully, the journey makes a good story that along the way introduces you to APIs and the empowering feeling that comes with them.

TABLE OF CONTENTS

BABY STEPS TOWARDS APIS AND DOCUMENTATIONS

DOCUMENTATIONS

When I started to study computer science, everyone was euphoric about the first hands-on coding course. Personally, I had no background in programming at all. I felt that everyone else in the room was way more experienced. What I didn't know: They were just as clueless as I was. At least most of them.

The course was about creating and controlling a Lego Mindstorms robot. After coming up with teams that had to be maintained during the course, as you can imagine, people were rushing with their ideas and implementations. It was intimidating for people without any background in programming. Even more when you are an introverted person. After all, you had to come along with your peers during the next months in the course. You didn't want to be the one who couldn't contribute anything at all.

A handful of older students taught the class in successive sessions controlling the robot. It happened often that they referred to an API when people asked questions. Referring to the API became the default answer.

However, I wasn't aware of what they meant with an API. Even when I finally asked for an explanation, the answer didn't help me at all. It was too abstract at this time. Obviously, it didn't help to read the Wikipedia article about APIs. I missed something concrete.

Initially, I thought the API is sort of a manual, or documentation one would say, to read about the functionalities of the robot. Because when the teaching students referenced the API, the learning students would always open up this cheatsheet of commands.

I was taught to be wrong with my assumption. The documentation only describes the API: How to use it and what to expect in return when using it. The API itself is the programming interface of the robot. The kind of things you need to execute in your code that something happens: class methods, functions. In fact, the API was the interface of the robot: How I could make it move, read the light sensor and play audio like the theme of Star Wars.

In a future course, I learned that a documentation of an API is not mandatory. Often, you can only read the source code that is behind an API to understand its functionalities. In one course, we collaborated with a Spanish university. We had to translate the documentation for a library from Spanish to English before we were able to use it. Even the source code of the library we had to use was in Spanish.

PACKAGES, LIBRARIES, MODULES - WOOT?

Eventually, I found out that these functionalities, that I used to control the robot, are somewhere hidden inside packages. The word package was used in a range of synonyms. They would have been referred as modules or libraries. It was difficult to untangle this abstract mess.

After all, these functionalities were grouped somewhere yet hidden. I couldn't see how these were implemented, yet it was sufficient to call a function to make something happen.

I am not sure when I had this infamous "aha" moment, maybe not even during the Lego Mindstorms course, but at some point I grasped that these packages, that I imported to access functionalities, were libraries.

At a later point during my university time, we were introduced properly to Java. Suddenly, a lot of things from the Lego Mindstorms course made sense. Even though no one mentioned APIs anymore.

In this course, we were introduced to the paradigm of object-oriented programming paired with syntax and patterns in Java. There were these packages again. They could be public, protected or private. To be fair, yes, they can also be package-private. But teaching these is not my goal in this article. The point is that these are access level modifiers in Java. They give you permissions to use functionalities from external places to the source code. Functionalities from classes. Functionalities from packages that bundle multiple classes. Basically they described the access level of the API. But nobody used the term API to describe those things in a practical use case. They were access level modifiers.

MEETING THE $

No, it was not about cash :) It was about jQuery. The first time I used jQuery was in a web development course teaching JavaScript in HTML. I must admit that the $ object confused me and I still wonder if others feel the same when they meet the object the first time. Wouldn't it be easier to simply name it jQuery instead of $? I have often mistaken the $ for a language specific syntax rather than an object coming from the jQuery library. That's why I didn't perceive jQuery as a library in the first place, but as well integrated thing in the JavaScript language.

The course went straight into jQuery to manipulate DOM nodes without using the native browser API at all. Yes, the browser has an API as well. To be more specific, there are

different APIs. For instance, one of it is the API for the DOM. You can use the document object to access your HTML, to traverse through it and to manipulate it. By substituting the native DOM API with jQuery in an early developer's career, no one ended up learning the basics.

Again, similar like the Java packages, I didn't perceive these functionalities of a library like jQuery or the native DOM as APIs at this point in time. But they would end up as two more concrete fragments to the greater picture of APIs.

ONE COURSE TO RULE THEM ALL: DISTRIBUTED SYSTEMS

After two years of computer science, I was still not convinced that I wanted to continue studying. I learned different theoretical and practical things but never got hooked. I was missing the bigger picture of it. Instead, I devoted most of my time to video and audio things.

However, there was this one course that changed everything for me. It connected all the dots. Suddenly it made all these practical things I learned previously relevant, because I could use them altogether.

The course was about distributed systems. It was empowering to get to know all the benefits that you get from a distributed system. Before, I was clueless about the possibilities in programming. However, this course changed everything because it gave me empowering moments. It opened up opportunities.

The course introduced us to different theoretical and practical things to enable distributed systems. Distributed systems are systems that are connected in a remote way. They can communicate to each other. In the course, these systems were often referred to as web services.

Suddenly, I could have a web service communicating to another web service. Even though both were physically not at the same place. One web service accessing another one would be the client. The consumed web service would be the server. But the best thing was that a client web service could be a server for another client web service again. That was one of this empowering moments I experienced. It would be a web of functionalities distributed somewhere yet accessed by APIs.

The main focus of the course was SOAP that is a protocol on top of HTTP. It allows the communication between remote web services. I hope that the course changed the content

communication between remote web services. I hope that the course changed the content by now, because SOAP seemed already to be dying back at the time. I am not even sure if that is true, but I never saw a SOAP powered web service after my time at university.

Fortunately, another 10% of the course taught REST. It is not a protocol like SOAP, but an architectual style that uses HTTP. It doesn't reinvent the wheel and uses standardized technologies to enable web services communicating with each other via HTTP.

But REST wasn't supposed to be the silver bullet. Whereas SOAP would have been used to expose functionalities, REST was intended to expose resources. Resources that you might know as a list of todo items, authors or comments that would be fetched from a server. Nowadays, RESTful services, services that follow the REST paradigm, are often misused. For instance, it can happen by exposing functionalities rather than resources. In most of the cases they don't follow all the principles of REST. They are more RESTish than RESTful.

But let's leave this topic for another time.

In my course, when the final decision had to be made in which paradigm my team of two would implement a web service, we fortunately decided in favor of REST. In modern web applications, RESTful services are commonly used as a standardized way to implement client-server-communication.

IMPLEMENTING A DISTRIBUTED SYSTEM: THE AFTERMATH

Did I mention that distributed systems were empowering? Did I mention that it connected all the dots of practical things I learned before?

Suddenly I was able to create remote services that expose functionalities or resources and could be consumed from other services that were physically somewhere else. In our team of two, we built a client-server-architecture that was powered by Java on both ends. We used Wicket for the client application and Jersey to establish a REST interface, an API that could be consumed from the client, for the backend application.

That's not even the whole truth. In the beginning, we used plain Java to connect client and server via HTTP. Eventually, we figured out that there were librares in Java that solved this issue already. We ended using the Jersey library to avoid the boilerplate. It was one of the times when you had to implement something the hard way, experiencing the problem on [illegible] and had the possibility to substitute it by a solution that was already out there

your own, and had the possibility to substitute it by a solution that was already out there.

The solution, a library, would be accessed by its very own API again. A library, with an API, to build an API.

Because these web services were decoupled by using a standardized communication channel, they didn't have to be Java on both ends. That was another great thing to know about. Suddenly I would be able to connect a JavaScript frontend with a Java backend application. Another empowerment along the way that was enabled due APIs.

Third-Party APIs

I didn't know that other platforms offered public APIs. However, it seemed like everyone else in the course, except for me and my partner, knew about it. It led the other teams to implement only a client-side application whereas my team built a full blown client-server-architecture. Now you have to see this in the context of our naivety yet curiosity. We spent a lot of time during these 4 months building the application. In the aftermath, I am grateful for it, because I learned a lot by building this application.

The principle of third-party APIs was another learning experience that had a lasting impact. Even though the client-server-architecture was distributed, I never thought about giving someone else access to the server by making the API public. However, again it was just mind blowing. It was that feeling of empowerment, being able to build functionalities that others could access remotely.

The Facade

I don't remember how we found out about the pattern, but suddenly we had a Facade in our server application. It is a programming pattern that groups functionalities in an interface object to make it accessible to the outside. In plain English, the implementation was just an interface that hid all the implementation details behind functions. However, it grouped these functionalities with purpose.

That was the point when we realized the RESTful service had an API. Similar to a browser with its native APIs, a library like jQuery, Wicket or Jersey, or your own implemented packages in Java. But this time it was an API to a remote entity.

A DEEP DIVE INTO REST WITH ROY T. FIELDING

Due to this impactful experience of distributed system, I devoted my Bachelor thesis to the

Due to this impactful experience of distributed system, I devoted my Bachelor thesis to the REST paradigm. After all, it was the topic that kept me hooked to computer science in the context of distributed systems.

I consumed everything I could find about the topic. It was still mysterious though. What does it mean that REST is an abstract architecture while SOAP is a concrete protocol? Everything about REST had to be laid down in front of me in order to process it piece by piece. I decided to read Roy T. Fieldings foundational thesis about REST.

There it was again: This paralyzing feeling of empowerment yet being overwhelmed. Principles in REST like HATEOAS and Idempotence hooked me completely.

A RESTful service is stateless? Wow, that means I would be able to use multiple services and distribute the server load among multiple instances. Another missing building block for my knowledge gaps. The biggest mystery was the authentification with a RESTful service. How should it be handled when the server is stateless? How does the server remember my session? I found out: It doesn't.

In my Bachelor thesis I made the straight forward attempt to compare different solutions that enable RESTful architectures in Java. I ended up comparing Jersey, RESTEasy and Restlet, libraries that enable you to build RESTful APIs, under the requirements of the offifical standardization JAX-RS. In addition, I compared their well doing under the light of the next generation of JAX-RS 2.0.

I guess every programmer would have said that it is a useless attempt to compare these libraries based on the standardization. However, personally it was a great learning exercise. It again taught me more in the fields of APIs. When comparing and implementing a RESTful service with three different libraries, you get to know the fine-grained differences. I was able to get to know the constraints of each library. I was able to get a feeling about good and bad API design. In addition, I could evaluate the constraints compared to an official standardization.

Leaving the topic of distributed systems, I want to thank my teacher, who had this lasting impact by lecturing distributed systems and for giving me the chance to write my Bachelor thesis about this topic.

LEAVING THE SANDBOX, FEELING EMPOWERMENT

The last years of my time at university, mobile development and single page applications

The last years of my time at university, mobile development and single page applications became a popular topic. Everything I learned before came together in both subjects. It was about accessing sensors of a mobile device, using libraries such as Ember to build a sophisticated single page application, designing own interfaces for Angular components, and gathering data from third-party platforms. I even built an own windows phone app in my spare time which consumed an open API of a popular platform. Everything by using APIs.

After my experience with distributed systems, I started to code in my free time. Suddenly I was able to build things. I was not limited to a dull sandbox anymore. Applications were able to communicate with third-party platforms, sensors and libraries. They could communicate with each other. Larger libraries turned out to be frameworks, yet they were consumed the same as a library by using an API.

I can only guess, but I assume that a lot of developers are unaware of using APIs all the time. Still, these things empower developers. Otherwise, developers would never be able to leave the sandbox. When leaving the sandbox, you can communicate with code from others, access their functionalities, retrieve resources from them or manipulate resources.

In addition, people are unaware when designing and implementing APIs themselves. It already happens when you implement a component in React or Angular. Be aware, that your peers are going to use it eventually.

I call it the unconscious act of using and implementing APIs.

HELLO, API!

APIs are the programming interfaces to applications. Simple APIs are thoughtfully and well designed. They follow constraints and don't monkey-patch or overload functionalities.

When learning programming, at some point concrete fragments, that are learned over time, create an abstraction. Suddenly, what Wikipedia said about APIs makes sense. The bigger picture unfolds in front of you. However, it takes time and concreteness in the fragments over the course of learning. In the case of an API, you suddenly become aware of the interoperability of systems and the contexts of using APIs:

- the DOM API in the browser
- the sensors in your smartphone
- the remote web service
- the library or framework in your application

- the library or framework in your web application
- the packages in your Java application
- the component in your React, Angular or Vue application

Everyone should be aware of it. Everyone is unconsciously an API architect. Eventually, others have to use your functionalities. Design and implement it thoughtfully. Keep it lightweight, simple to use and with clear constraints. That's what makes your API durable over time.

APIS ARE CONCRETE: THEY HELP TO LEARN YET TO TEACH

Through my journey at university, I learned programming with APIs. Distributed systems was the subject that kept me learning programming. By using APIs with concrete examples, be it the browser API, a third-party library or a remote RESTful service accessed by its API, you can use these concrete examples to learn. I found it highly empowering, leaving my sandbox in order to learn programming. You get the real world content from others to experiment with.

After reflecting on this topic of APIs, I try to use them to teach programming. Consider a third-party platform that has an API to access its resources. Rather than leaving students bored by having them to push around arbitrary data, you can give them access to real world data. I use this concept of teaching programming with third-party platforms often. Yes, you run into problems with changing APIs, but you still empower your students. I use this principle in my book to teach the fundamentals of React.

. . .

In the end, I hope that the article didn't come along to blame the university, teachers or anyone. In the contrary, the journey supported me a lot to digest all the smaller building blocks. When learning programming everything seems to be so far away. When teaching people you need concrete examples that are not buried under abstractions. Still, a beginner can easily get overwhelmed by all the noise burying the important topics. You can feel paralyzed by it. But you can fight this effect by using concrete examples to teach and learn programming. APIs are a great example to empower students.

Show Comments

KEEP READING ABOUT WEB DEVELOPMENT›

HOW TO TEST AXIOS IN JEST BY EXAMPLE

Every once in a while we need to test API requests. Axios is one of the most popular JavaScript libraries to fetch data from remote APIs . Hence, we will use Axios for our data fetching example...

WHY I STOPPED USING MICROSERVICES

I've always been fascinated by APIs. In fact APIs, distributed systems, and web services are the reason why I learned to code . When I started my first job as a junior frontend developer, it wasn't...

50.000+ readers.

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development with JavaScript
- ✔ Tips and Tricks
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

Online Courses
Open Source
Tutorials
About me
What I use
How to work with me
How to support me

Contact Me Privacy & Terms