JavaScript on the Client

This lesson explains how JavaScript is an important technology to use from the client side development perspective.

Client-Side Development:

JavaScript has been clearly the language to go to when it comes to client-side development. Let's stop for a moment. What is client side?

> ***Client-side*** development focuses on writing code that runs in the browser of the end user.

Back in the 90s and early 2000s, JavaScript was a little toy language that could *animate* elements on a static website. These animations were called *dynamic behavior*.

Animation and Rendering:

> ***Animation*** in computer science is the act of moving an object on screen. During the move, the features of the object may change, which means that the animated object may be drawn in different color, different size, or in case of three-dimensional animation, we might see the object from a different angle.

> Displaying an object on screen at a given point in time is called ***rendering***. Animation can also be defined by continuously rendering an object on screen, while its features, including its position, may change.

A Brief History of JavaScript

Twenty years ago, JavaScript was not taken seriously in the software development industry. Many "developers" had no idea what they were doing.

and they just copy-pasted code snippets to add some dynamic behavior to

their site. Often times, these snippets clashed with each other, because they used the same variables.

AJAX Programming:

JavaScript became more popular, once *AJAX* programming (Asynchronous JavaScript and XML) started spreading. In 2006, the `XMLHttpRequest` object became standardized, meaning that you could submit a request to the server and wait for its response. Once the response arrived, you could display the results *without reloading the page*. This worked like magic on websites because before this technique became wide-spread, every interaction with the server required a full page reload. As a page reload may take seconds, getting rid of the need for reloading pages was a big achievement.

The JavaScript community was growing, but we had another problem at hand: in order to write code that works in all major browsers, we had to write different code for different browsers. Sometimes the situation was hopelessly confusing. Therefore, you had to be a **domain** expert to write JavaScript code.

jQuery:

Not for long though. In 2006, another important milestone happened. The library `jQuery` got released with the slogan *write less, do more*. Instead of writing tens of lines of code to make sure you covered all browsers, jQuery gave you single line commands to achieve the same result. You didn't have to be a serious software developer to write JavaScript code. You just had to include jQuery, and your problems were solved.

Unfortunately, `jQuery` created another problems. We could use the slogan of Scarlet Spider, the evil clone of Spider-Man: *all the power, none of the responsibility*. Lowering the entry barriers meant that people who didn't care much about software engineering, also started writing code.

Single Page Applications:

At the same time, the idea of *single page applications* and *client-side rich web-applications* became mainstream. Before AJAX and jQuery, almost everything was on server-side. The task of the server was to assemble a web page, and send it to the client. The server created HTML, CSS, and JavaScript code, and

send it to the client. The server created HTML, CSS, and JavaScript code, and the client only loaded that one page. This changed when the application logic was moved to the client. This meant the application had to be loaded once, and multiple megabytes of JavaScript code was responsible for rendering the part of the application the client was interested in. This step was made possible by **Moore's law**, because the performance of the end user's computers doubled every few years.

> ***Single-page applications*** receive all the HTML, CSS, and JavaScript on page load, and the client can navigate between different views of the application without informing the server. The client communicates with the server with AJAX calls, and receives data without the need to reload the page. This way, single-page applications became faster, and as easy to use as a desktop application.

Maintainability Issues:

The main problem was that people calling themselves software developers, used their hacking skills to create bad quality software. People soon realized that some jQuery knowledge limits the size of the application they can develop. Beyond a certain size, they encounter maintainability issues.

> ***Maintainability*** is the act of making a software solution easy to develop, modify, scale, test, and debug. If you write ten lines of code, you need a different approach than writing a million lines of code.

JavaScript Frameworks:

The answer to the maintainability issue was the emergence of JavaScript frameworks. Frameworks made it possible to write maintainable code in a structured way, without the need to reinvent the wheel. Frameworks enforced some well known best practices from the field of software-engineering. In order to understand these best practices, the entry barriers got raised again to a healthy level.

> A ***framework*** structures your code and runs your application. When

> using a framework, the framework is *in control*. This means, the framework calls your code.

Library:

> A ***library*** is a set of functionalities that you can call to perform a given task. A library is not a framework, because as long as you use a library, you are in control, the library does not call your code.

In the early 2010s, literally, everyone wanted to write a framework. A new framework got created literally every week. Creating a framework means that you reinvent the wheel, because you solve a well known problem in your own way. If you want to build a portfolio, don't even think about building a framework, because it does not make sense.

Fortunately, we are past the time when the **churn rate** of JavaScript development was so high that you had to learn a new framework or library every month. Today, it is quite clear that you use React or AngularJs or VueJs, and React seems to be winning the battle. There is a big community behind the main frameworks, because React is backed by Facebook, and Angular is backed by Google.

The frameworks were also great, but developers had one more problem. The frameworks and libraries were so much beyond the capabilities of JavaScript that JavaScript became a burden. The language was already quite old, and we needed an update. This update was **ES2015**, or using its old name, **ES6**. Many innovations made it to ES6 from different sources such as `jQuery`, some `frameworks`, some functional programming libraries like `UnderscoreJs`, and some languages like `TypeScript`.

Quick Quiz!

1 Which of the following is not a framework?

Conclusion:

Since ES2015, there was no turning back. Every year, some updates made it to JavaScript, and these updates also ended up in my course, JavaScript in Practice: ES6, ES2017 and beyond.

We can conclude based on the history of JavaScript that there has never been a better time to be a JavaScript developer. In many companies, you can make more money than a C++ developer or a PHP developer. The challenges you deal with on a daily basis are interesting, and more importantly, you often get immediate feedback on your efforts by seeing the effects of your changes in your browser.

Summary:

- JavaScript was first a toy language for creating dynamic behavior
- AJAX made it possible to update websites without reloading the page
- jQuery lowered the entry barriers to programming
- Frameworks and libraries forced us to use well-known design patterns in software engineering
- ES2015 and beyond: many patterns became a standard part of the language