JavaScript Module Systems

Author : Sajith John Sam
Email : sajith.sam@ayatacommerce.com

What is covered?

- Real-Life example
- Modularity in coding
- Intuitive Modularization
- Wrappers to Rescue
- IIFEs as better wrappers
- Here comes common-js
- CommonJS pitfalls
- And finally, the robust ES module system
- Other module systems & references

JavaScript Module Systems

Real-Life Example

Advantages

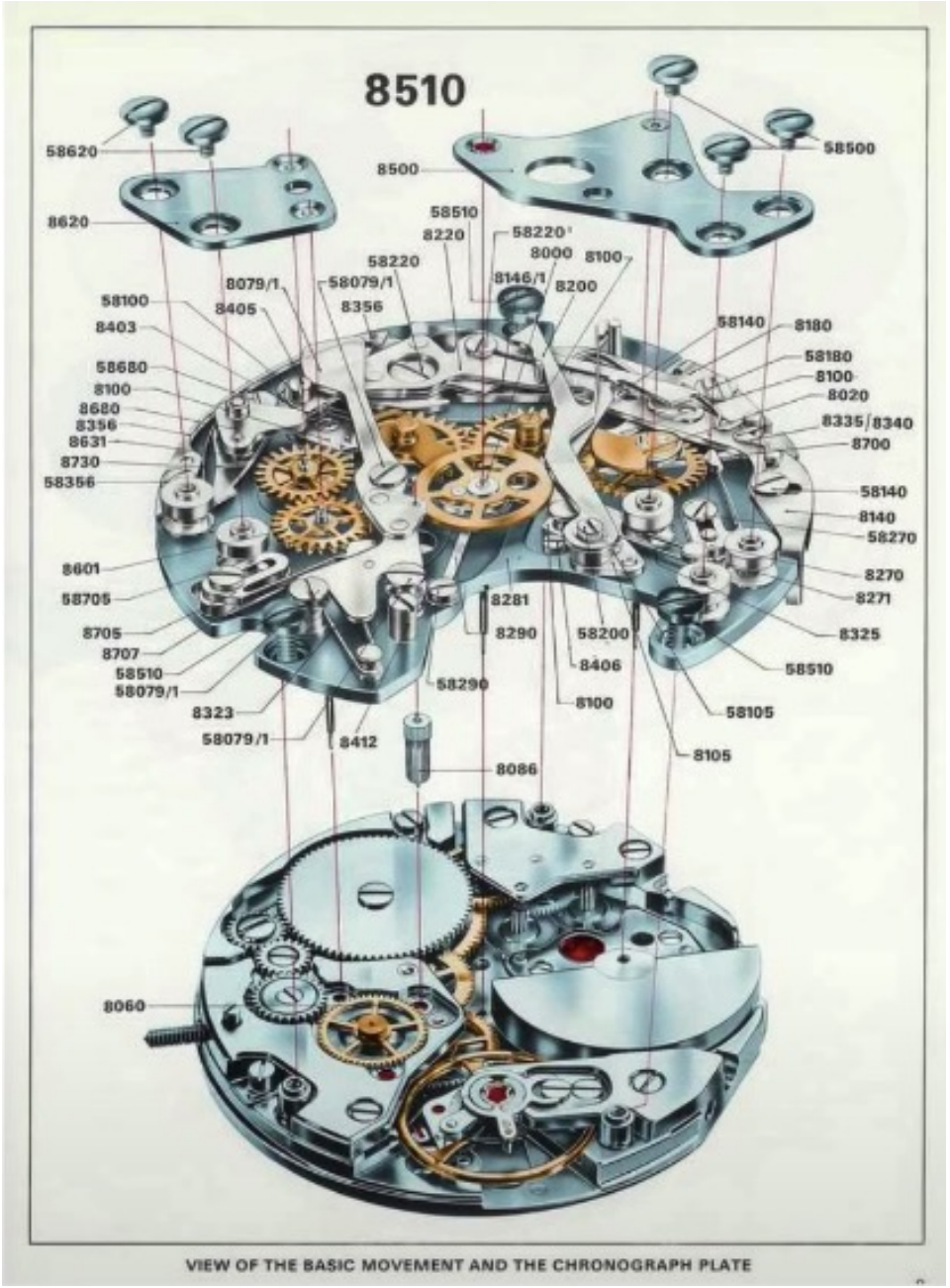
Reusability

Composability

Leverage

Isolation

Organization



JavaScript Module Systems

Modularity in code

How this can be related to code?

Just as the watch was designed, we *should* design our software separated into different pieces where each piece has a specific purpose and clear boundaries for how it interacts with other *pieces*.

In software, these *pieces* are called **modules**.

JavaScript Module Systems

Intuitive Modularization

Intuitive Modularization

Why not put stuff in different files?

<code>demo</code>

Any *<script>* tags in our HTML pages will cause properties to be added to the global *window* object.

The result is *namespace pollution* which can lead to unwanted side effects.

So, this is *not* the right approach, now what?

JavaScript Module Systems

Wrappers to rescue

Wrappers to rescue

Namespace pollution can be dealt with using *wrappers.*

<code>demo</code

But we can refactor things for better using IIFEs

ayata COMMERCE

JavaScript Module Systems

IIFEs as better wrappers

IIFEs (Pronounced as EE-Fi)

The term IIFE stands for
Immediately Invoked Function Expression
If a function is written just to be executed once, we can replace it with an IIFE.

<code>demo</code>

IIFEs introduce other problems:

- Namespace pollution is not completely dealt with.
- Order of execution matters now.

There should be some solution, here comes Common-JS

JavaScript Module Systems

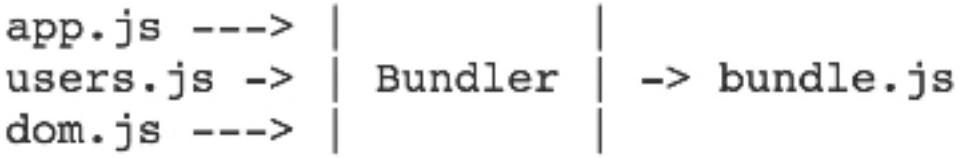
Here comes Common-JS

ayata COMMERCE

Common-JS

The Common-JS module system has the following features/caveats

- File-based
- Explicit Imports
- Explicit Exports
- Default module system in NodeJS, so very popular
- Not very popular with browsers, so it requires a bundler like ***Webpack or Browserify***



The Common-JS group defined a module format to solve JavaScript scope issues by making sure each module is executed in its own namespace. This is achieved by forcing modules to explicitly export those variables it wants to "expose" to the universe, and by defining those other modules required to properly work.

- Webpack Docs

<code>demo</code>

JavaScript Module Systems

Common-JS Pitfalls

Common-JS doesn't solve all the problems

The Common-JS module is good but not perfect.

- No async loading of modules (Browser's memory stack is hung when a module is loaded)

- The *require* statement is a function and so it doesn't give you the feel that you're importing something. This often leads to a flawed design.

- There are other downsides which doesn't warrant a discussion in this thread.

JavaScript Module Systems

ES Module System

ES module system

The Common-JS module was good but not perfect.
The ES module system is better – close to perfect

Inbuilt

- async loading
- import/export syntax
- Better control over the dependency injection

<code>demo</code>

JavaScript Module Systems

Other Module System & references

It is not all over

There are two other module systems which you will come across in your JS lifecycle.

- Universal Module Definition (UMD)

https://github.com/umdjs/umd

- Asynchronous Module Definition (AMD)

https://requirejs.org/docs/whyamd.html

My references

Tyler McGinnis' awesome explanation of the module systems

https://www.youtube.com/watch?v=qJWALEoGge4