TailwindCSS
TailwindCSS is made by the authors of Refactoring UI

It is a utility CSS framework, so before we get into tailwind we have to explain what that is.
The main principles of utility css are composition and descriptive classes, so instead of something like

```
.subVideoBoundingBox {
height:100px;
width:100px;
display:flex;
}
```

You have multiple classes that define height, width, display like

```
.h-100{          .w-100{          flex{
height:100px;    width:100px;     display:flex;
}                }                }
```

and you would use them like class = "h-100 w-100 flex"
The advantage is that reusing css classes reduces the overall file size, reduces complexity, and you don't have to context switch out of what you're doing to edit the css.

Additionally, since the classes are self descriptive, someone that's familiar with the framework can glance at the markup and know what it looks like, instead of looking up classes in the css file.

Of course, then you end up with markup that looks like:
class=" hidden md:block items-center px-2.5 py-1.5 border border-transparent text-xs font-medium rounded-full shadow-sm text-white bg-blue-400 hover:bg-red-200 focus:outline-none focus:ring-2 focus:ring-offset-2 focus:ring-red-200"

It looks really ridiculous but you have to try it to believe in it

This style of development is to build it out first, and extract the repeating parts into new components/classes, instead of trying to plan out which parts will be reused from the start.
It works better with component frameworks.

So why Tailwind in particular?
First of all, rolling your own utility classes is a drag and it won't be something someone else can reuse easily, it's better to use a framework. (Especially if you don't really know what you're doing)
Tailwind is probably the first implementation of utility classes that does it really convincingly.

The biggest selling point is the documentation for tailwind, which is really excellent, to the point where I would recommend beginners to start with tailwind instead of vanilla css. There are tons of things about CSS to actually learn, and tailwind has the most important ones documented, and it's easy to experiment and learn what does what, since it's a matter of changing one line on the fly instead of looking up a pile of classes in the css file and editing them.

For the really powerful css features like selectors, animations, queries, etc. you can still just write your own css and use it along with tailwind, you can also edit the configuration file to extend it with custom utility classes.
Also! It makes responsive design really easy because it comes with breakpoints like xs: md: lg: and so on, which is kind of tiring in regular css

If you’re interested in reading more have a look at the creator’s blog
https://adamwathan.me/css-utility-classes-and-separation-of-concerns/

On WebRTC

WebRTC is a p2p communication protocol, it has a bunch of specifications to enable p2p communication.

The ICE framework is what allows peers to connect to each other, they discover ice candidates and send it to the other peer over the signalling server. It is important to note here that the signalling process not specified in the webRTC standard, it could be anything from calling them with a list of ice candidates to sending it by mail, usually we have a dumb server just forwarding it, it's very cheap.

You might think, aren't websockets a p2p protocol, well not really because browsers don't and shouldn't accept arbitrary websocket connections, and usually users will be behind a NAT with no public address, along with firewalls and other things. It is possible to hack around all this, with things like NAT hole punching, etc., but using the webRTC protocol we don't have to worry about all this.

WebRTC specifies a Session Traversal Utilities for NAT (STUN) server, which handles the discovery of the user's public ip address and connection details, the peer pings the server and sends this information to the other peer. This is also very cheap, in fact there are plenty of public STUN servers that can be used for free.

If the user is not accessible at all due to security rules etc. webRTC specifies a Traversal Using Relays around NAT (TURN) server, which forwards packets to the other peer. This is not very cheap; the difference between a TURN server and a regular media server is that the TURN server just blindly forwards data.

If the ICE exchange process is successful, we have a p2p connection between both browsers that doesn't rely on anything else.

On handling multiple peers

The simplest way to handle multiple connections is a full mesh network, i.e. every peer connects to every other peer once, this uses a lot of resources for each peer but what can you do. I hoped that someone would have come up with a way to route partial data to each peer or something like that, but haven't found anything.

Otherwise you would go back to a regular server client setup, where each client sends data to the server once, and receives the data it needs once; webRTC is still preferable over sockets for media connections here because it is explicitly designed for media connections.

Our implementation in the project is p2p video/audio streams, and arbitrary data like messages and game information. This is sort of the 'hello world' of webRTC, and covers most of the basics and cases that would need to be handled. The version of the project running locally does not use any stun server, because it doesn't make any sense and is also very confusing, we're using peerJS so that specifying a public stun server will only take a few lines of changes.