The Complicated Way You See Patient Data: a Discussion on EHR Front-Ends for Doctors

Erkin Ötleş
February 6th, 2022

Topics: Healthcare Information Technology, Electronic Health Records, Computer Science, Software Engineering, Software Architecture, Clinical Informatics, Tech Support

I have a love-hate relationship with electronic health records (EHRs). This relationship first started in the early 2000s at a high school sports physical and has significantly outlasted my high school soccer career. Eventually the relationship turned serious and my first job out of college was for an EHR vendor. My thrilling life as a support engineer at Epic Systems Corporation was cut short by my silly decision to pursue an MD-PhD. After years of being on one side of the software and data-stack I transitioned to being a “user" for the first time. While not totally naive to all of the issues surrounding modern EHRs this transition was still pretty eye opening.

I believe a significant subset of these issues actually stem from a general lack of communication between the engineering community making these tools and the medical community using them. One of my goals in pursuing the MD-PhD was to hopefully help bridge this gap a little bit. As such, I’m usually game to play tech support on the wards and I like explaining how the software we use works (or doesn’t). I also like translating what we do in medicine to the engineers that will listen. Basically I’ll talk to any crowd that will listen (maybe this is why I went into academia 🤔).

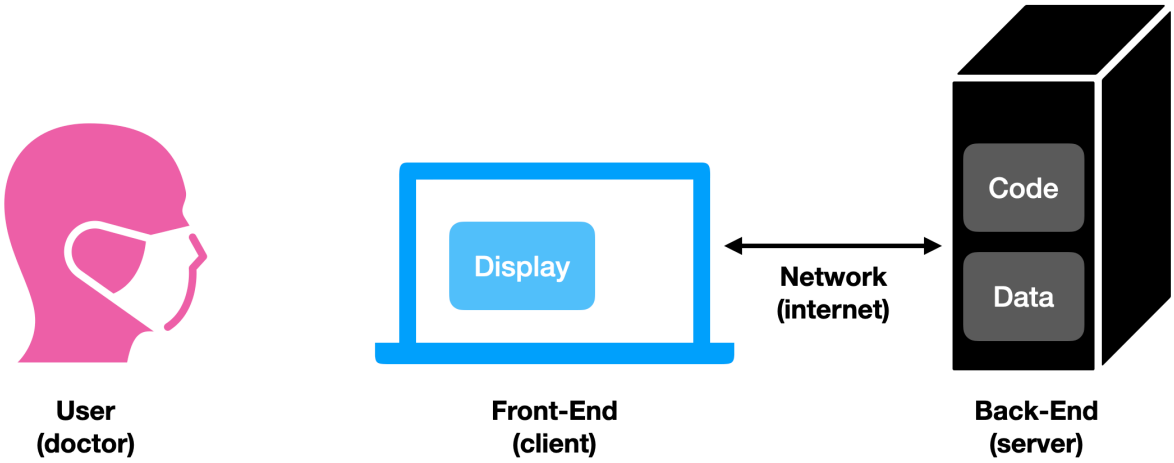
This post is inspired by a tech support call I fielded from Jacob, one of my med school classmates. Jacob was about to take an overnight call shift and his computer was displaying the EHR in a manner that made the font so small it wasn’t readable. I walked through some potential settings in the EHR that could be affecting what he was seeing, but everything we tried came up short. Eventually Jacob texted his co-resident and they told him to try modifying a Citrix Receiver setting, which worked. My singular focus on the complexity inside of the EHR instead of the complexity AROUND the EHR led to my tech-support failure. The complexity around the EHR will be the focus of this blog post.

Concurrently serving an EHR to thousands of physicians, nurses, and allied health professionals across health systems is a big task. This task, like most other software tasks that involve interacting with users, is broken into two big components, with a *front-end* and a *back-end*. [1] This is an over simplification, but the front-end is everything that a user interacts with and the back-end is all the other stuff that needs to exist in order to store and transmit data used by the front end. You’ve probably been the beneficiary of this division of labor even if you’ve never written any code. Twitter, Facebook, Youtube, and Gmail all use this approach.

Let’s take Gmail. The front-end of Gmail is all the code that needs to run on your laptop (or phone) in order for Gmail to show you your emails. The back-end of Gmail is all of the code that Google needs to run in order to store your emails, send your outgoing emails, and receive your incoming emails. In order for you to see your emails Gmail’s front-end and back-end need

to communicate, they do this by passing messages back and forth. A similar setup is employed with EHRs. The front-end of the EHR is what shows you the lab values of a patient. The back-end is what ultimately stores those lab values along with notes and other data.

This separation of front-end-back-end makes engineering easier as it decouples the information presentation functions from the functions that actually run the service. This allows engineers to upgrade the look and feel of a website without having to worry about redesigning the way the site interacts with a database. Ultimately this separation enables specialization and efficiency. One set of engineers can focus on making the front-end look good and another set can focus on making the back-end run fast. As long as these engineers trust each another they work efficiently by focusing on their own domains.



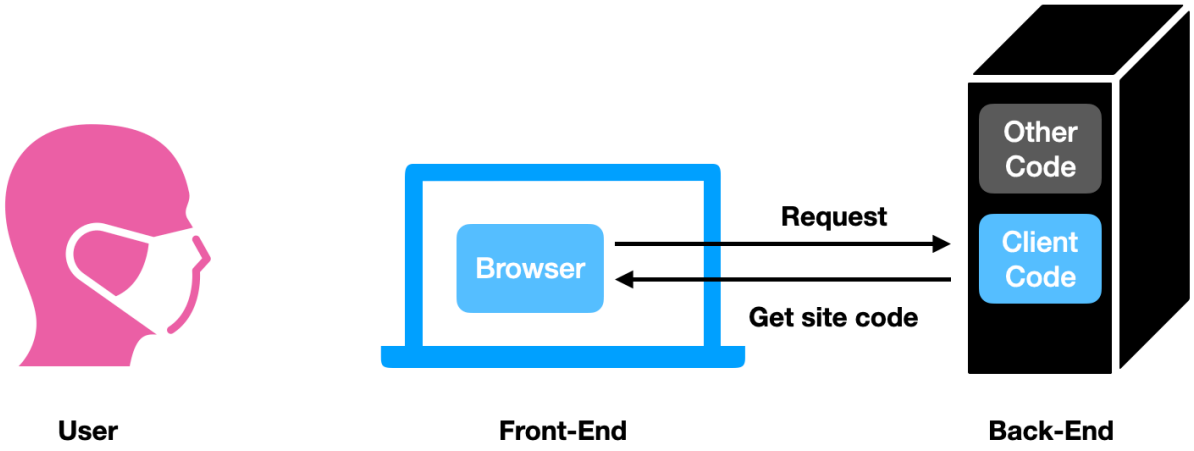
Front-end back-end division of labor. Both the front-end and back-end have code but they specialize in different tasks. The front-end code, also known as the client, displays information, and facilitates user interaction. The back-end code and services, also known as the server, stores data and contains code to facilitate the system's non-user functions such as networking with other systems. The front-end and back-end will communicate by sending messages to one another over a network.

The software that makes up the front-end is often known as the client. The amalgamation of everything on the back-end is often known as the *sever*. [2] Its a little facetious to talk about a single client and a single server, because most well-known projects might have multiple clients and many servers. However, its not too far off from the current state that most EHR users are familiar with. For this post we will keep our focus on the front-end/client side of things.

Let's stick with Epic's EHR system. The client most everyone is familiar with is Hyperspace, which can be found in clinics and hospitals all over the US. [3] I don't know if there's any data on this but I'd hazard a guess that the Hyperspace client accounts for over 95% of the total time users spend with Epic's EHR. (That guess is based on my own usage time as a med student.) Although I mainly used Hyperspace, I would occasionally check up on my patients using Haiku or Canto. Haiku is a client designed for smartphones (there are apps for both Android and iOS) and Canto is a client designed for iPads. Additionally as a patient I use MyChart to access my own medical records. All of these clients are designed with different goals in mind and provide differing access to clinical information and workflows.

Each one of these clients needs code in order to display information and facilitate user interaction. Usually clients accomplish this by having code that runs on the machine the user is using. For example the code for Canto is downloaded on my iPad. When I click on a patient's name on Canto code executes (that code was probably written in the Swift language). That Swift code may change what is displayed on the screen and may also send or receive messages from servers. It may do any number of additional things, but the primary user interaction and communication tasks are handled by code that is running on my iPad. This set up is pretty similar for Haiku, the only difference is that its running Swift on my iPhone instead of my iPad. MyChart and Hyperspace are different. There's a superficial difference, which is that they are clients that don't run on iOS/iPadOS devices. But there's a deeper difference, which is how the user's device gets access to the client code.

That's the tricky part. Its also related to Jacob's tech issue. Getting access to the Haiku or Canto client is fairly straightforward. They are apps that you can download from the Apple (or Google) App(Play)Store. You download the code, its on your iDevice, if Epic wants to push an update they can upload a new version to the AppStore, and Apple will take care of updating the code on your iDevice. MyChart and Hyperspace are different, very different. One can think of a couple reasons why they might be different. But in my mind primary driver of the differences is time. All of these clients were introduced slowly over time and each one follows the primary client deployment paradigm of the time they were developed in. Walking backward through time in a very simplistic manner: the AppStore was a big deal when it came out in 2008, it upset the web-based paradigm of the early 2000s. The 2000's web-based paradigm itself had taken over from the locally installed application paradigm of the '90s. MyChart follows the web paradigm and Hyperspace follows the locally installed paradigm.



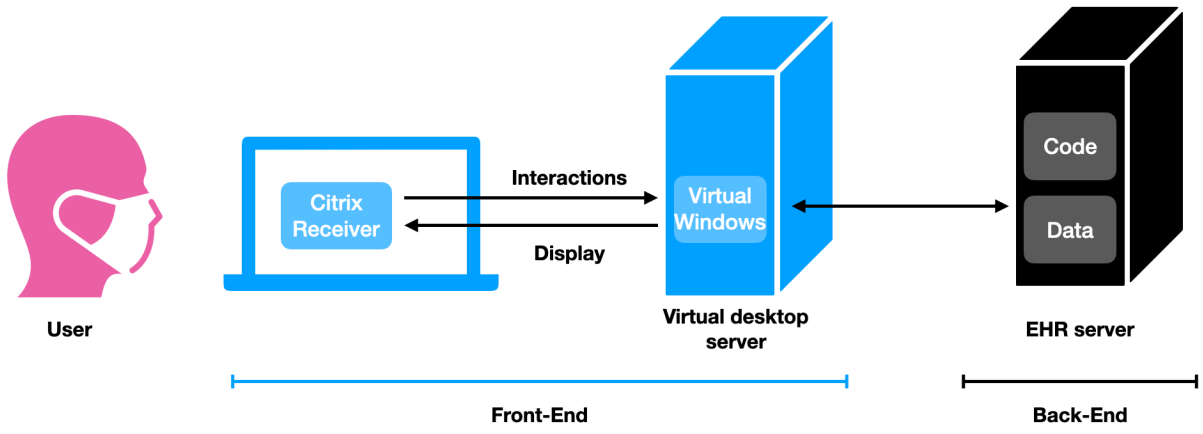
Web paradigm for client deployment. When the user visits a website their browser requests code from that website's server. That front-end code runs inside of the browser and enables the user to interact with the site's back-end services.

The web paradigm is sort of cool because the idea is that client code is sent to the users device just-in-time. It is also how all websites work. When you you tell your browser to go to your favorite website, the browser gets a bunch of code from that website. That code package is made up of HTML, CSS, and Javascript and tells your browser what to show and how to interact with the back-end. Since the client code is requested when you visit the site, front-end developers do not need to worry about pushing updates to multiple devices. They just need to update the server that serves the front-end code. From that point on all users that visit the site

will get access to the latest and greatest code. Pretty slick, because you don't need an Apple-like middle man to keep everyone's client code up to date. MyChart for the most part works like this. Its not quite as straightforward because MyChart is tied to each health-system that uses it, so the updates from Epic will need to go through them in order to be seen by patients.

Finally we get to Hyperspace. Hyperspace, by nature of being Epic's most capable client is also its most complicated client. The internal complexity of Hyperspace was what I was thinking about when I was troubleshooting with Jacob. Despite this internal complexity Hyperspace has the potential to be the simplest client to deploy. As mentioned above, it uses the locally installed paradigm. Every child of the 90s should be familiar with this paradigm; you find an a program you want to use from the internet (or get a cd), download the executable, run through then installation process (🧙). Then you use the downloaded program to your heart's content. That's the paradigm that Hyperspace was designed for. In the early 2000s, at the time of my high school sports physical, that was the paradigm that was used. When my doc launched Hyperspace, he was running code that was installed on computer sitting in the room with us. When a new clinic was to be set up all of the computers going there needed to have Hyperspace installed on them. When Hyperspace was updated all of the computers in all of the clinics and wards needed to have their software updated. Additionally, installing and running hyperspace locally on all these computers meant that all the computers needed to meet all the requirements needed in terms of RAM and compute power.

As you can see, installing and using Hyperspace entirely locally is problematic. The deployment management perspective alone is headache inducing. And what if people want to access the EHR from home? Users would need to install Hyperspace on their own machines? And need to keep them up to date? Forget about it! The solution to these headaches is brilliant in a way. Hyperspace needs to run on a windows computer, but that computer doesn't need to physically exist in the clinic as long as the people in the clinic can virtually access that computer. Enter virtualization.



Virtualization of the front-end. The front-end code that displays the EHR to users does not actually run directly on the user's computer. Instead it runs on a virtual computer. This virtualization enables a great deal of IT efficiency.

Virtualization, specifically desktop virtualization is best described by Wikipedia: "desktop virtualization is a software technology that separates the desktop environment and associated

application software from the physical client device that is used to access it." [4] What it enables is moving all of those individual computers (and the Hyperspace client) to virtual Windows servers. Then all the computers in the clinic need to do is to connect to those servers. Those virtual Windows servers will then present the whole desktop experience to the users. Maintaining virtual Windows computers is a lot easier than maintaining physical Windows computers. Updating software on those virtual computers is a lot easier too. In the late 2000s Citrix released software that enabled businesses to have virtual desktops and for other computers to connect to those virtual desktops (Citrix Receivers, AKA Citrix Workspace App). [5] If packaged properly, you won't even notice that you've launched into another computer, you will just see the application you are interested in using. This is what currently happens with Hyperspace.

So Hyperspace went from being installed locally on the computers in clinic to being installed locally on a virtual Windows computer that you access from clinic (or home). The way you access the Hyperspace client is through another client, the Citrix Receiver. This Russian nesting doll setup has added some complexity but greatly also greatly simplified deployment headaches. Using virtualization is pretty cool because it allows locally installed clients to be deployed in a manner analogous to web-based deployment. You end up trading off one type of complexity (managing lots of local installations) with another (maintaining virtualization), but on the whole it's a good trade for IT departments.

What of Jacob's issue? Well it turns out it was a Citrix Receiver issue. As a client Citrix Receiver takes your mouse and keyboard inputs sends them to the server running Windows and Hyperspace virtually. This virtual computer returns what should be displayed and Citrix Receiver displays it. Some time before Jacob called me, Citrix Receiver had updated and asked if Jacob would like to update his resolution settings, he had inadvertently said yes. This in turn made the fonts on Hyperspace appear really tiny. Reverting that setting helped return the Hyperspace display to normal.

When Jacob told me about the fix and how it involved changing a Citrix Receiver setting I kicked myself. Its the one part of the system I would never think to check. It was a good reminder that there's a lot of complexity built into every part of the system that serves our patient records. While I spend most of my time thinking about other parts of the EHR this bug was a good reminder to not forget about the humble client.

Erkin

Acknowledgements

I'd like to thank John Cheadle (LinkedIn) and River Karl (LinkedIn) for reviewing this work prior.

Bibliography

1. *Frontend and backend - Wikipedia*. 2022; Available from: https://en.wikipedia.org/wiki/Frontend_and_backend.
2. *Client–server model - Wikipedia*. 2022; Available from: https://en.wikipedia.org/wiki/Client%E2%80%93server_model.
3. JEFF GLAZE jglaze@madison.com, -.-. Epic Systems draws on literature greats for its next expansion. 2022.
4. *Desktop virtualization - Wikipedia*. 2022; Available from: https://en.wikipedia.org/wiki/Desktop_virtualization.
5. *Citrix Workspace App - Wikipedia*. 2022; Available from: https://en.wikipedia.org/wiki/Citrix_Workspace_App.

