Guitar Tracker Dashboard Design

Add Headings (Format > Paragraph styles) and they will appear in your table of contents.

About This Dashboard

I built this hobby project to track my progress learning the classical guitar and to explore the new Shiny for Python framework. This section of the dashboard documents the design and development process including requirements, wireframes, high level design, data engineering, data integration, and visual development. I'll also pepper in challenges I ran into and solutions/lessons learned for future dashboard projects using Shiny for Python. If this sounds like fun, please keep reading!

I have included the full source for this application via my github repo (link at bottom) in case you see something useful in here that you'd like to carry into your own project. While this application by default reads from a live PostgreSQL database, I've included a local SQLite version of the data in the repository. Simply cloning the repository and running the dashboard should have it automatically read the on-disk database. More on this is in the Repository Structure section.

I pulled out some parts of this dashboard that were reusable and started tracking minimal reproducible examples in a separate git repository: Python-Shiny-Examples. I'll refer to some specific examples in this writeup.

Requirements

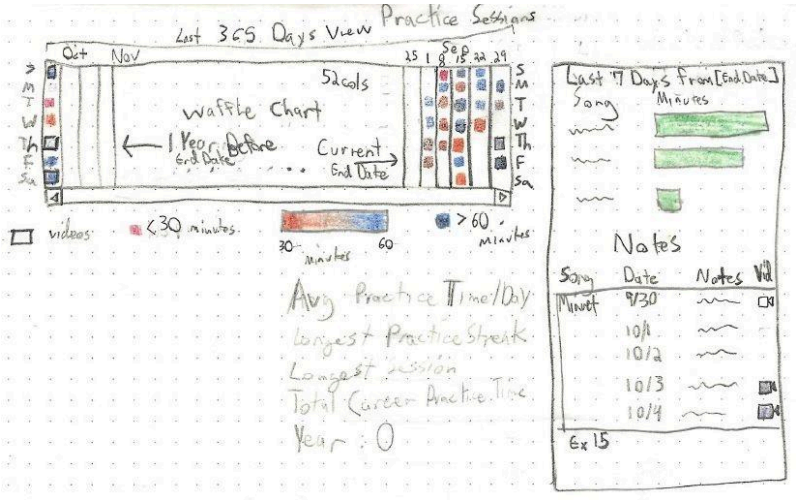
When I first started guitar lessons and practice sessions, I kept a journal writing down for each practice session how much time I spent on each song, what I focused on and what I found difficult. After a few weeks of lessons and practice, I started thinking about using a dashboard instead of a notebook to communicate with my teacher, and track progress over time. Given that I was looking for excuses to jump into the Shiny for Python framework, this seemed like a great use case. I set out to answer answer these questions visually:

- Practice Sessions:
    - How much time do I practice each day/week, and what songs am I spending my time on? Specifically for this, I also wanted to be able to post practice videos on some days and have those available
    - Over the past week, what parts of each song did I focus my practice on and where did I struggle?
- Career:
    - For each song, how much time have I spent, Learning Notes, Achieving Tempo, Phrasing, and Maintenance.
    - Separately from songs, how much time have I spent practicing certain exercises?

    - Thinking about the "10,000 hour rule" to become an expert, how many hours have I spent actively playing guitar since I started?
- Inspiration/Goals: *(Stretch Goal)*
    - What songs have I heard performed that I found inspiring and would like to play some day?
- Acoustic Arsenal: *(Stretch Goal)*
    - What guitars am I playing on, and how many hours have I spent playing each guitar? Someday I may want to upgrade from the entry level guitar I currently use.
    - What strings do I have currently installed on each guitar and when should they be replaced based on how old they are and how much I play on them (like tracking miles against running shoes)?

Visual Wireframes

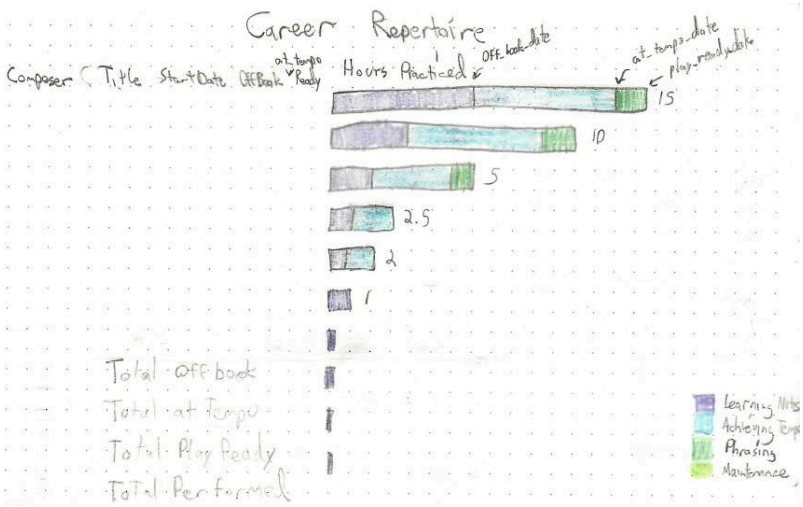
These questions helped organize the project into logical sections that eventually became tabs on a nav panel. I had some ideas up front of how I wanted to visualize the data and drew out on paper a few wireframes of visuals that I'd like to see.



I thought it might be neat to see my practice sessions for the past 365 days represented as a waffle chart like the GitHub/GitLab commit graphs. As I started drawing it out on paper, I realized that it was starting to look like a guitar fretboard, so I decided to add a headstock to the right end of the visual. Additionally, I drew out a section that I'd display on the main page that showed my teacher what I focused on for the previous week during my practice sessions. At a minimum I wanted this to show the songs I focused on over the previous week and total time spent on each. I wanted a table view for this part where I could present my notes on each session for the past week grouped by song and then sorted by date.

For the career tab, I knew immediately that I wanted to show a stacked bar chart with the time spent in each "stage" of learning for each song (Learning Notes, Achieving Tempo, Phrasing, and Maintenance). I figured that I could define a few milestone dates for each song and when I met them, I could define the stages as the time between each. The bars themselves would be ordered by total time spent working on songs. This makes me wonder though if eventually all

the songs I learn over the course of years will have a very long maintenance bar and barely visible sections on learning the songs proportionally.



When considering the Inspiration/Goals metrics, I had this image in my mind of a plaque with plates on it that each represent a song I want to learn. It would form the bases of a sort of inspiration wall.

The idea here would be to have each song I want to learn someday that I've heard featured as a plate on a wooden plaque background. Each song can have multiple arrangements of differing difficulty levels, so really it's the arrangement that I want to identify as a goal. When the user clicks on the plate, a body of text opens up on the right with details about the song itself, and then a section for each arrangement that I want to learn.

Clair de Lune is an example of a song that I love hearing, and the arrangement that I enjoy the most is far too advanced for my skill level. But it's a goal nevertheless. Then I found a simplified arrangement that is still really hard, but in the realm of possible for me. I want both arrangements and their sheet music links to show when I click on the Clair de Lune plate.

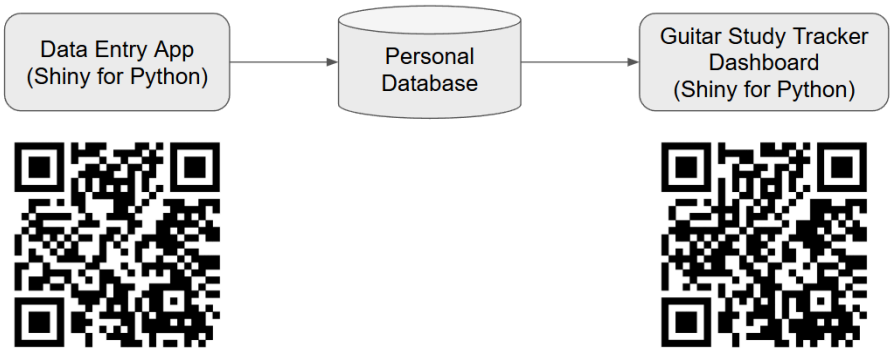
High Level Design

Data Sources

Normally when I work on a data project for a customer who owns the data, I'll stop here and ask a customer for the data sources and start to look at the structure to:

1. See if they have the correct data and enough data to show or derive the insights they are looking for, and
2. Gauge the level of effort to ingest, join, and transform the data into the appropriate shapes for the visuals they want to see.

However, for this project **I am** the data owner and there is **no data!** Truthfully, this is the ideal scenario because I have total control over every aspect of the data. I can realize any visual idea I have in mind as long as I structure the data appropriately right from the start when it's entered. I settled on this basic data flow for my project from data input to dashboard:



I'd build a front-end application that allows me to enter all the table data I need at the correct granularity level, store it in a SQL database and then write the dashboard application to query that same database, extract the tables, join them and visualize the insights. With most of the major domain, process, and technology requirements known, It's time to decide on a technology stack for the project.

Technology Stack

I already mentioned that a large part of this project was focused on exploring the Shiny for Python web framework, but I did some research and landed on these major technologies/services:

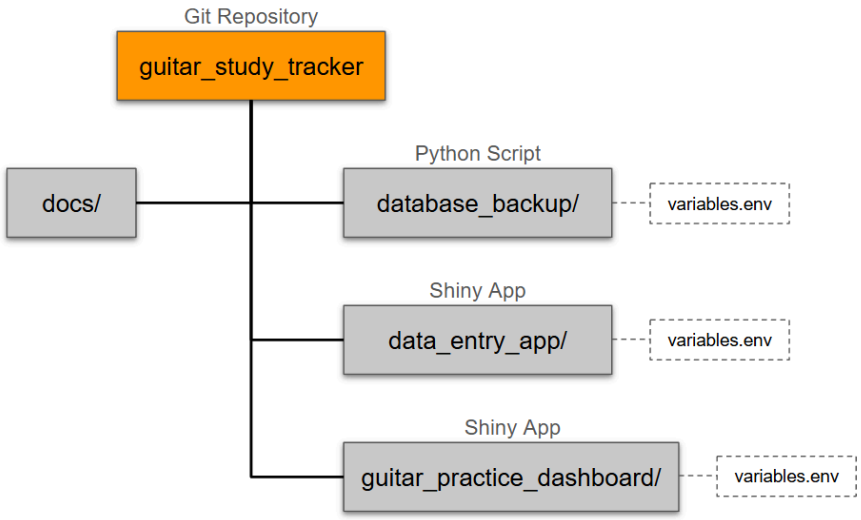
- **Python/Shiny for Python** - Open source python web framework for HTML/JS/CSS generation and for server logic - my excuse to do this project. 🙂
- **Plotly** - This is an open source visualization framework for Javascript with bindings for Python and R. I selected this framework over other python frameworks like plotnine and matplotlib because it's interactive by nature - visuals can be configured to respond to

click events, make accessible the data represented in those selections and allow the developer to affect other parts of the app based on that selection.

- **Supabase** - This is an online database hosting service with a small free tier that fits my hobby project perfectly.
- **Shinyapps.io** - I was debating between shinyapps.io and connect.posit.cloud. I would have picked connect.posit.cloud, but read that it is still in an early release stage and isn't intended for any kind of scaling. While I don't expect this app to get a ton of traffic, I decided to deploy it to shinyapps.io hoping it can stay there a while.

Repository Structure and Offline Data Access

This project actually has three applications in various locations in the repository. Each application also has multiple files. The repository structure is as follows:



Each of the code folders on the right contain either a shiny app or python script. The applications all have different jobs but they share in common that they connect to the PostgreSQL database only if the variables.env file is found in their folder. This file is not tracked in the git repository in any of the folders, but it exists on my development environment, and it exists in shinyapps.io abstracted from the user. If the repository is cloned fresh, the variables.env file will not be present. The code will instead by default connect to a local SQLite database file in the same directory that includes previously cached data (from an output of the database_backup application). The local database IS tracked in the git repository. This makes it possible to clone the repository and play around with the dashboard code while having access to a copy of the data in almost the same format as it is housed in PostgreSQL.

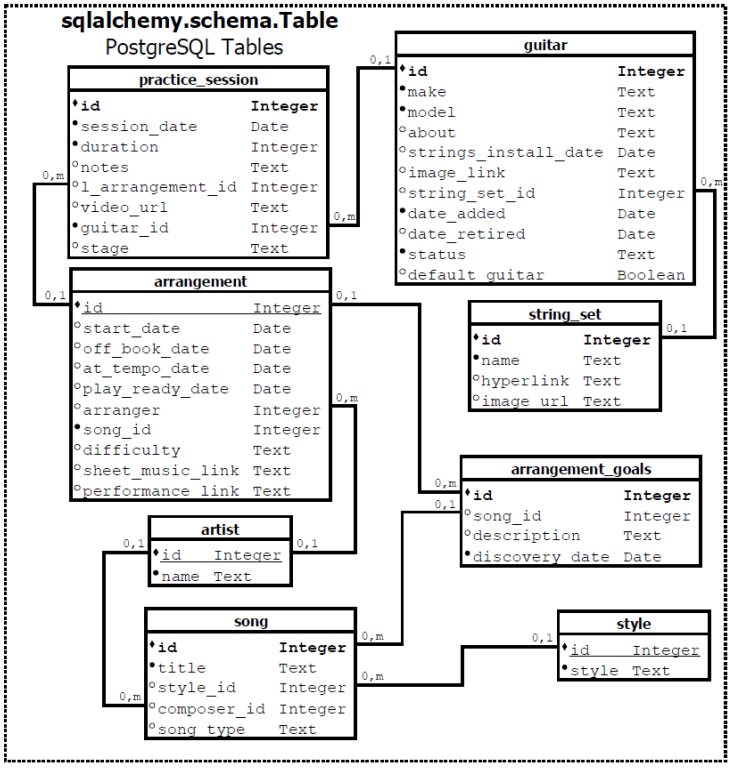
There are a few differences between the PostgreSQL and SQLite database that are apparent in the Data Entry App where adding new records to the local dataset doesn't populate the primary key of each table 'id'. PostgreSQL does that on its own since it understands that 'id' is a primary key. Working on the dashboard with cached data should appear in parity with PostgreSQL.

Data Engineering

This section focuses on designing the database and data input application. For most of the visuals on the Sessions and Career tab, I would need to produce a dataset where each row represents a unique session_date/arrangement combination and included the number of minutes practiced, and notes, and which guitar I used. For the Goals tab, I would need a dataset where each row represents a unique arrangement with information about the song (a single song can have multiple arrangements). Then for the Acoustic Arsenal tab, I'd need a row for every guitar in the arsenal that includes information about the strings installed. I would also need to perform some table calculations with practice session data to determine how much many hours I've played the guitar and how many hours the I've played the guitar with its currently installed strings.

Schema

Putting it all together, I came up with the following database schema:



I got this set up in Supabase and then set up a data entry app using python and sqlalchemy to read from and write to the tables. I decided to set up two main accounts, one that is read only and one that is read/write. The database hostname, port information, schema, and read-only credentials are stored in the variables.env file that is NOT tracked in my git repository but is copied directly to shinyapps.io when I use Posit's rsconnect command line interface to publish.

Data Integration

I used the SQLAlchemy Object Relational Mapping (ORM) library to handle database read/write functions. This meant I needed to define each table format in ORM friendly objects before querying anything. I kept it simple and just extracted the raw tables, leaving the joining to python/pandas rather than work out all the processing code in SQL - I'm stronger in Python anyway. However, I housed the database code inside its own classes to keep things portable. Since I had a number of tables to manage, I created each tables data entry screen as a module. This allowed for encapsulation of ui code for each specific table's lookup-resolved view while sharing common elements like the basic table form UI and button functionality. I've since become more capable at using modules, and I'm not entirely sure that this object oriented method of working with modules is better than just using the shiny modules themselves. A full class diagram for the data entry app is located in the repository in the docs folder.

There were a few notable challenges I ran into during this part of the project. Building a database front end data entry tool in Shiny was humbling at first, but after working through some of the initial challenges

Challenge: Early on when I started working with Shiny I tried to place module code in loops (particularly the server code) thinking I had to run it every time I wanted to access the same page in my multi-tab panel app. This led to run-away reactivity where I end up clicking a button for example and then seeing it's effect run multiple times. I thought that I might be having problems with the reactive objects in my server function being interdependent, but actually it was that I launched my server function more than once. So if I had a server function that was listening for an action button with id="my_btn", then after I clicked on it, the server function for that button would run as many times as I invoked the server itself.
Lesson Learned: When working with modules to handle iterative ui/server functionality (like adding a unique button to every row in a dataframe), I learned to ensure the ui/server module code is ONLY DEFINED ONCE for each module namespace_id. Essentially, don't let there be more than one of the same ID in the DOM, and be sure that the server code for each module is only invoked/defined once. **Note:** I ran into this same issue later on when working on the dashboard, but for a different reason.

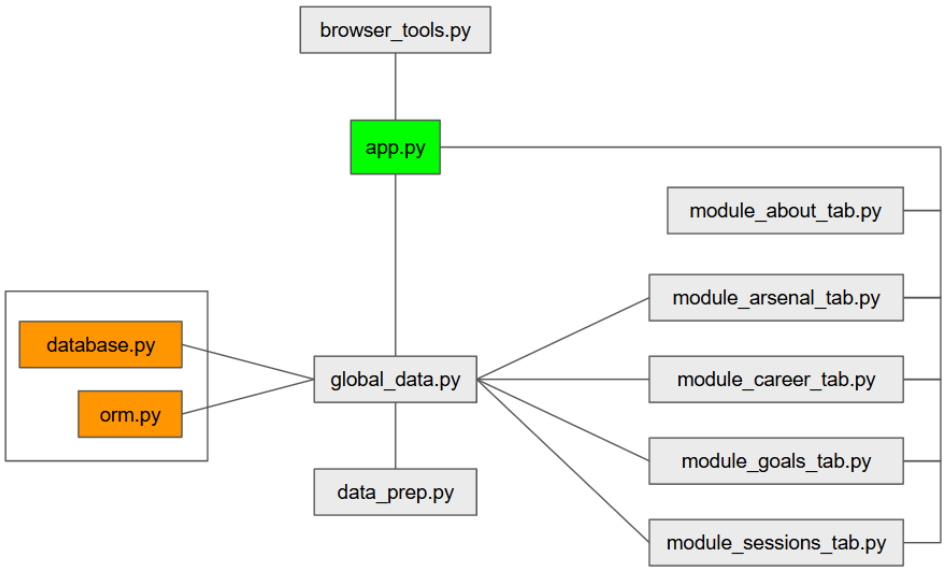
Challenge: The full set of tables is queried only once when the data entry app loads. If I updated a table to add a record, only that table updates when I refresh the data, none of the other table views knew about it because their processing code wasn't run again. I have to wait for the session to expire or restart the application to fully refresh the data of all the related views that access that table I updated.
Lesson Learned: I didn't go back and fix this issue in the data entry app, but may in the future. Certainly if I were to do this again, I'd set up a central singleton/subscriber pattern for data processing in order to allow the various table modules that relate to a specific table that was updated to subscribe and gain access to updated data any time one of the tables is adjusted.

Ideally this would be a reactive singleton - because shiny handles the subscriber/notifier functionality for us with reactive objects. I took some parts of this and applied it to the dashboard - such as a singleton datastore.

Main Dashboard

I set out with a few technical/management goals for this part in addition to building a dashboard. I wanted to split my codebase up into as many modular components as I could both to reduce the size, but also make it so that if I were working on a team I can break off sections of the dashboard and split the work up among multiple people. Splitting into multiple files also helps reduce the chance of git conflicts.

With that said, here is the file structure of the dashboard:



The data integration code (database.py and orm.py) files are highlighted in orange because they are the same file in each of the three applications in this repository. Of note in this structure is the global_data.py file that interfaces with both the database and the global data prep. I built this in to limit duplicate calculations of the same data in multiple modules, and because the application is small enough that global data doesn't become a disaster. Within global_data.py is a non-reactive singleton class that is instantiated from each module to pull the data they need for their visuals. Several modules needed the same processed data, so this was an easy way to feed it to them.

Debugging Shiny and Logging

Shiny is heavily asynchronous. This means that certain blocks of code are kicked off and run in the background, allowing the program to continue execution of other tasks. Debugging this kind

of code sometimes requires additional things like the built in debug mode, loggers, profilers, and plain print statements to understand the program's flow order when something doesn't seem to be working correctly.

One of the tools that I've seen with RShiny that I do hope to see in the Python version soon is the reactlog. RShiny has this awesome profiler that runs while an RShiny app is running, and builds a visual reactive dependency graph. I think that it's essential toward rooting out and eliminating interdependent reactives or runaway reactivity. I've dealt with a few of these situations while working on this project and in some cases I couldn't resolve it until I drew my own dependency graph stepping through the code. That exercise usually allowed me to "isolate()" the issues. 🙂

I also built a logger tool that helped a little with locating interdependent reactives by echoing out to the console when a function starts running, and when it ends. It's like a print statement with some understanding of its scope. The idea is that the programmer imports the logger and invokes it at various points of the program. This will cause the logger to output to the console when it's invoked and again when the function it's called from goes out of scope. Here is a small working example of the logger from the Python-Shiny-Examples git repository.

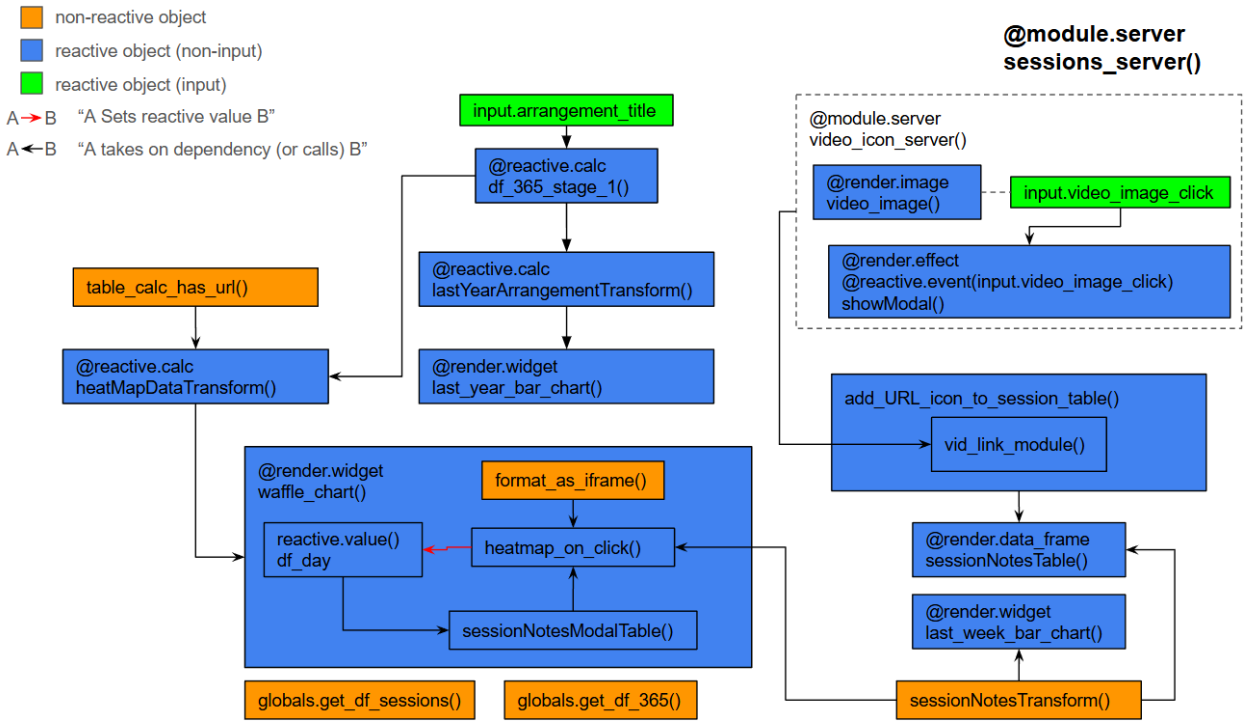
app.py Structure

The app.py is actually a really small file that defines the skeleton of a fluidpage navbar layout, defines a static footer that sits at the bottom of each screen with author information, and finally calls a shiny module for each tab, where it's ui and server logic will reside in its own namespace. One thing to note is that the goals tab has an extra reactive object passed to it (the browser resolution). That tab redraws itself based on the width of the browser viewing it (desktop vs mobile) and needs to know when the resolution changes.

The browser resolution isn't immediately available in Shiny for Python (that I could find as of Shiny v1.1). I put a javascript function inside browser_tools.py that is called the from app.py server function to query the browser resolution as (x,y), and save it to a reactive value called input.dimension().

I'll describe steps taken and notes for each tab below and also include a reactive dependency chart.

Practice Sessions Tab



This view becomes complex particularly with the waffle chart. This plotly figure contains a few extra functions. When a user clicks on a segment of the waffle chart (representing a single day), that causes df_day (a reactive value to be set with a dataframe of session data for that day. The code then generates a modal with a table view on it for session data for that day and then shows any linked youtube videos in an iframe.

While it looks suspiciously like there may be a reactive interdependency between heatmap_on_click, df_day, and sessionNotesModalTable(), df_day is only updated once when the user clicks on a waffle segment. Then its value doesn't change again until the user selects a different segment. This causes df_day to update once, and present cached data to its callers after that, stopping a reactivity feedback loop.

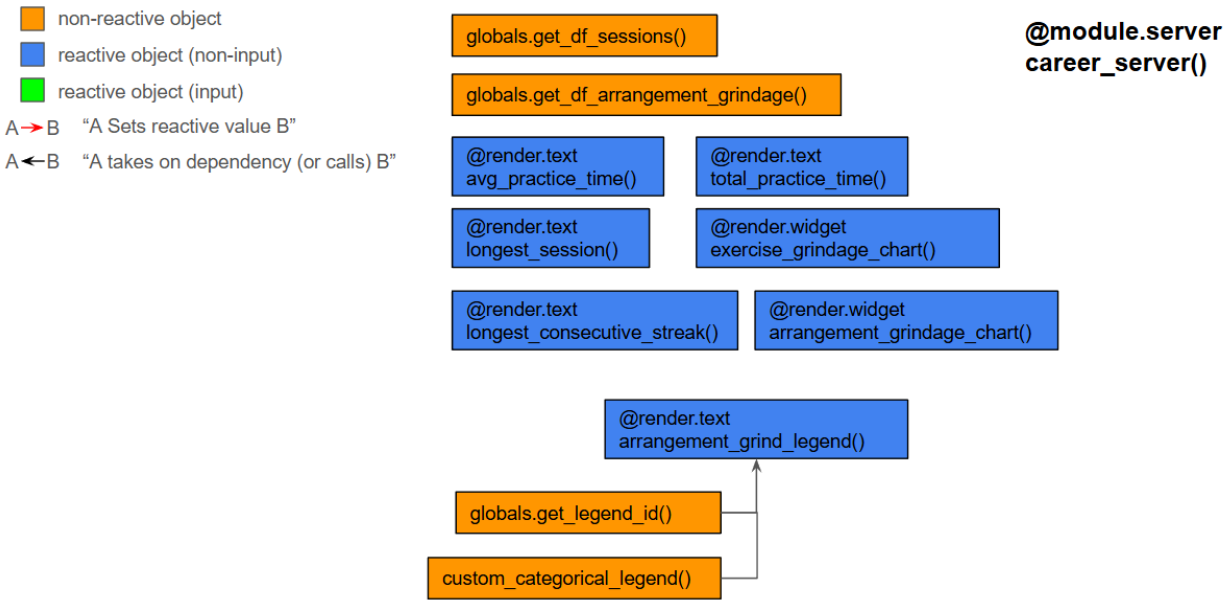
sessionNotesTable renders a dataframe of practice sessions over the last week. I have the ability to capture one Youtube recording per arrangement per session date in my data model. If the table has any URLs posted in the URL column, instead of showing those as an href that sends you away from the dashboard, I have it display a clickable video icon. When clicked, each video icon brings up a modal of the respective youtube video rendered in an iframe. This iteration over rows of the dataframe means to build modals specific to the row required shiny modules to accomplish. I called the module from the sessionNotesTable in a df.apply() function in order to add the row's id to the module namespace.

Challenge: I ran into an issue when trying to build this sessionNotesTable to render the video icon and modal. I foolishly built it as an add-on to a previously built function that caused an unintended side effect. Originally, before I built the sessionNotesTable and before I linked sessionNotesTransform up to the waffle_chart(), I built the sessionNotesTransform() non-reactive function to process data ONLY for last_week_bar_widget(). When I wanted to add in the sessionNotesTable with its video icons, I modified the sessionNotesTransform() function to add the vid_link_module reactive function that launches shiny modules for each URL in the table. Then I had the reactive sessionNotesTable() call the sessionNotesTransform(). I had forgotten that last_week_bar_chart() was still calling the non-reactive sessionNotesTable (which now housed calls to a server function). This means that the sessionNotesTransform was getting called exactly twice each time it was called, once by the last_week_bar_chart() and once by the sessionNotesTable(). It also means the shiny module was launched twice. If the user clicked on the video icon for some row of the sessionNotesTable, they would see the modal render twice (one on top of the other).
Lesson Learned: I asked for help in the Shiny discord channel and one of the Posit Devs kindly came to the rescue. After I drew out my reactive net and realized what I did wrong, he helped explain the situation to me in fewer words than I used above. The lesson here was to be careful to separate values and side effects. I initially used the sessionNotesTransform() as a value providing function that returned a dataframe. When I introduced the vid_link_module function to that function, it took on a side-effect in addition to returning a value. The solution was simply to move the vid-link-module() function out of the sessionNotesTransform function and place it somewhere that only the sessionNotesTable would trigger it. I placed it in its own function add_URL_icon_to_session_table() that exclusively manages that side effect. *Looking back, I think this challenge is a pretty easy trap to fall into, especially for folks new to the framework*. I'll probably try to fall into it again at some point and hopefully remember to look back at this.

One thing about this view that I'd probably try to avoid in the future is having sessionNotesTransform be a non-reactive function. As it is, it runs three times producing the same output for each of its callers. It doesn't present a latency issue, but it's kind of wasteful. I'd make this a reactive calc in the future so that it calculates once and then provides cached responses to future callers.

Career Tab

I decided not to add any dynamic sorts or filters to this tab, so it just calculates its values once when the tab is loaded and never recalculates them:

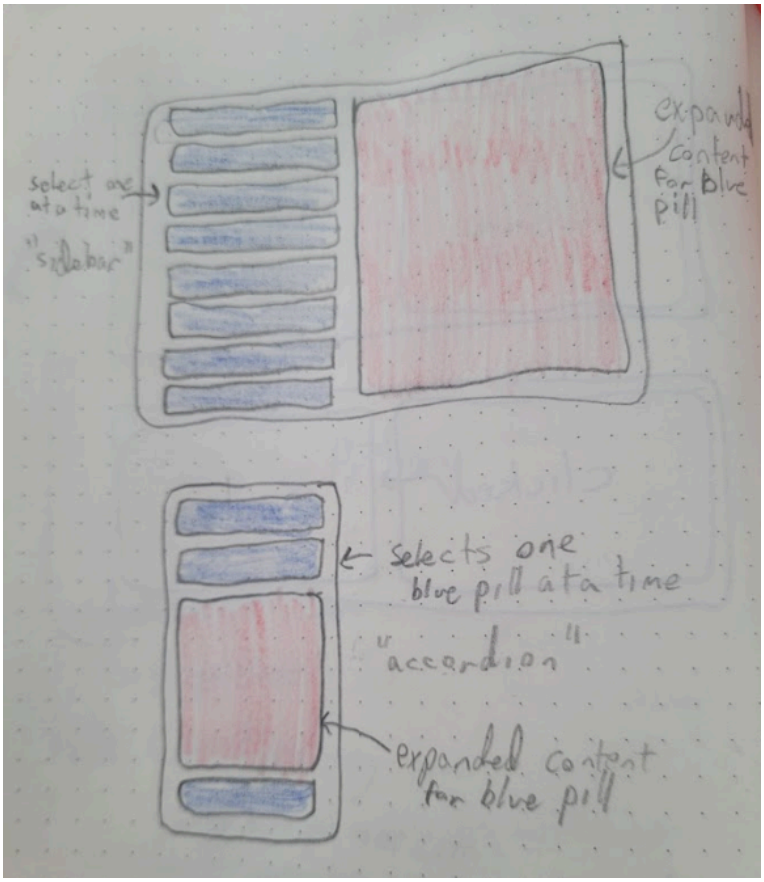


I may decide in the future to add a dynamic sort to the arrangement grindage table so that I can sort it by the total time plated (it's current settings), or change it to the most recent practice date. It'd be nice to know which songs I've learned but am falling out of practice on and might need to pick them up again.

For this tab, I fought a bit with plotly to format and place the legend just where I want it. It was doing undesirable things when I switched from desktop to mobile view that wasted a lot of space. I wanted to detach it from the plot entirely so I could put it anywhere, but was struggling to make that work. Finally, I hid the legend and created a custom data driven ui (arrangement_grind_legend) to look like the legend inside a flexbox styled div so I could place it where I wanted and it would be responsive.

Arrangement Goals Tab

This was the most challenging tab, because I wanted to explore a more DIY web layout but also make it responsive. It ended up being a good choice to leave for last. This was a workout in CSS, shiny modules, and also responsive design where it can be viewed on a desktop or mobile device and still make sense. I wanted the plaque with plates layout shown in the drawing earlier in Wireframes to be presented as a sidebar layout when viewed on a desktop where the plates exist on a sidebar to the left, and the content extends off to the right. However, if the user views this on a mobile device, I wanted to show the content as an accordion where the main body of content is shown between the plates when one is selected. I was after something like this:



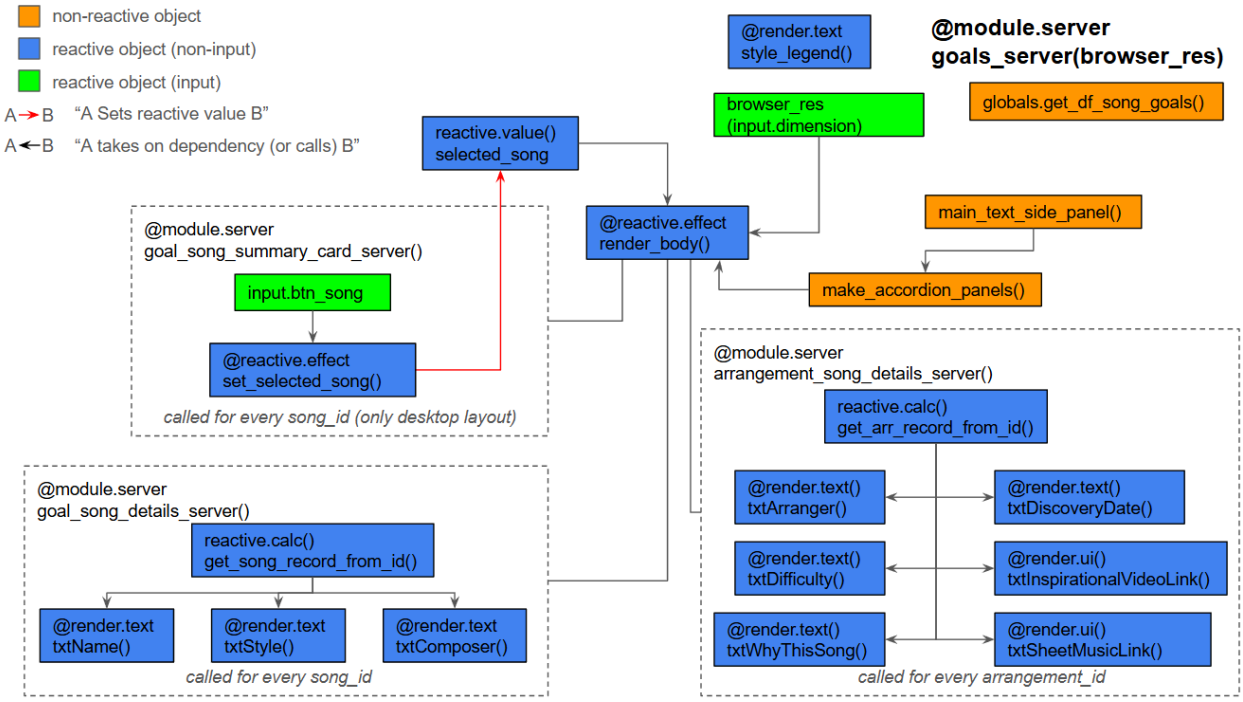
I initially tried to use CSS exclusively to handle the responsive nature, but decided eventually to switch between desktop and mobile view by having the server detect what the browser resolution was and redraw different UI objects for mobile or desktop using ui.insert_ui and ui.remove_ui.

I placed a javascript function into browser_tools.py mentioned earlier to get the browser resolution and then make it available to my server function as a reactive input object called input.dimension. It updates whenever the browser resolution changes Here is a small working example of the browser_resolution listener from the Python-Shiny-Examples git repository. Then I passed the input.dimension into the goal_tab module as a reactive value. This allowed me to use it as a reactive value in that module's server function and redraw the layout to desktop or mobile depending on the resolution.

The desktop view was probably the hardest to piece together because I ended up building the layout with divs rather than try to use the shiny sidebar layout. This made it possible though to style the wood background and the plates (which are action buttons with divs for their labels instead of just text) just how I wanted. Even still, I struggled with CSS styling to get all the right parts to do what I wanted. Too often I'd apply a setting to find that either the default bootstrap was overriding it, or simply that I applied it incorrectly. I started learning to use flexboxes here too which presented their own learning curves. I'm not really proficient with it yet, but I can tell

that the flexbox system in CSS is going to be a really powerful way to build layouts when I become more familiar with it and want to build something very custom in Shiny.

The reactive net here was also more complicated than other tabs, there are multiple modules at work here. Even still, this go around with modules was more straightforward after having worked through simpler examples earlier:



The module server function for this tab goals_server() starts off by launching the following server modules:

- For every song_id:
  - goal_song_summary_card_server(): This is for each plate when shown on the desktop layout ONLY
  - goal_song_details_server(): This is used to show the details of the song. It's used both in desktop and mobile layouts
- For every arrangement_id:
  - arrangement_song_details_server(): This provides details about each arrangement in a song. Some songs have only one arrangement identified, and some (like Clair de Lune) have multiple arrangements.

Those module server functions run in the background and respond to their respective ui elements by id value. render_body() analyzes the browser_res reactive value (remember this comes from input.dimension on the base server function), and determines if it will render for desktop or mobile. It then removes and inserts the appropriate ui elements because they use the same ui element ids. This ensures that no ui element ids are ever duplicated.

Acoustic Arsenal Tab

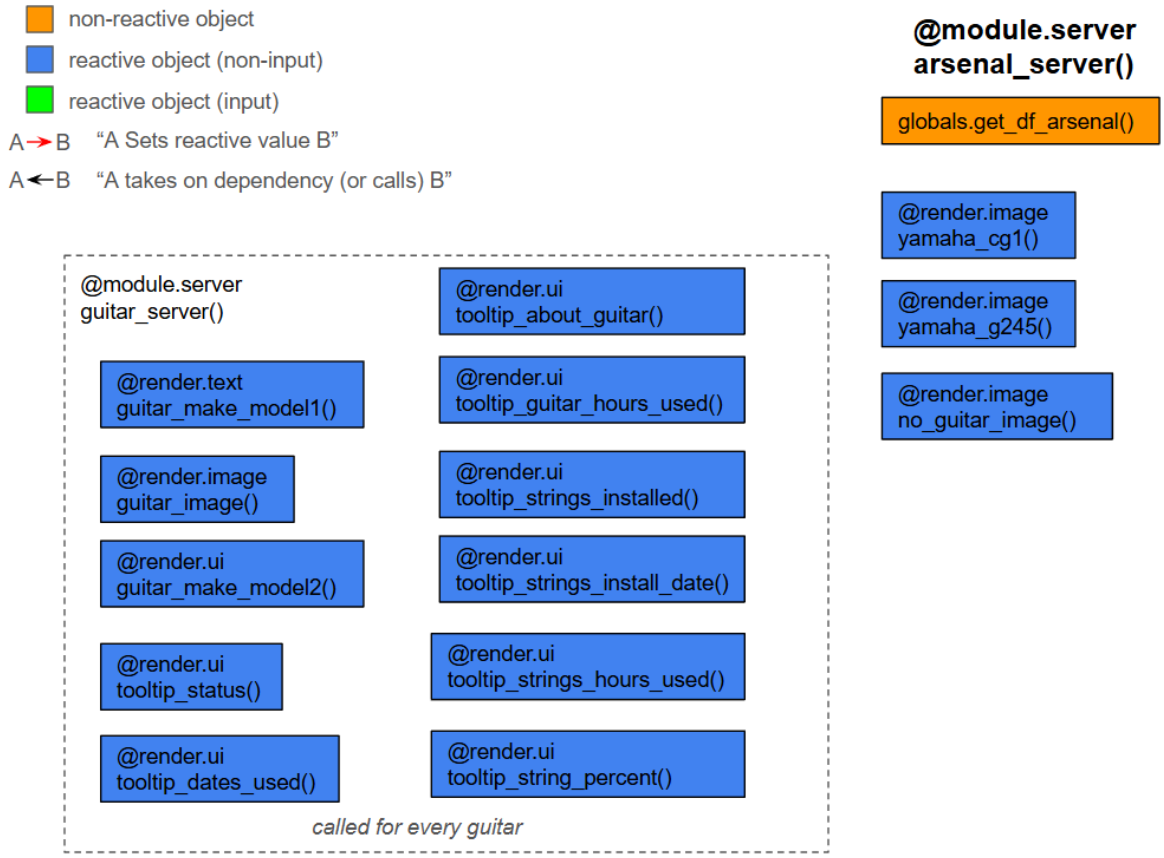
This tab is minimally interactive, and came together quickly. It will however require the most maintenance because I'll need to add an image and server function for each guitar that I use to practice on. This will be pretty infrequent though.

It shows an image of each guitar I've registered in the arsenal, placed in a flexbox horizontal with wrap turned on. Hovering over the guitar brings up a pop up dialog that shows information about the guitar itself, playtime spent on the guitar, how long I've had it, how old its strings are, etc.

There was a little bit of math involved, where I wanted to come up with a way to calculate the health in percentage of the guitar's strings, and extrapolate a date by which the strings should be replaced. I did some research on this and found that it's mostly personal preference when to change strings. There were some guidelines though that I saw a few times, like replacing strings after 40-80 playing hours, OR after about 4 months-time of minimal play. I built this into a model where it calculates a degradation value each day from both those methods and uses the largest, this cumulative degradation turns into a percent health.

$$h(n) = -\sum_{i=0}^{n}(max(kt_i, d)) + 100$$

where:
h(n) = string health at day n
ti = time played on day i (minutes)
k = time-based coefficient = 1/60 hrs
d = day-based coefficient = 1/112 days



About Tab

This write-up! It was produced using a simple shiny layout and lots of ui.markdown().

CSS Considerations

Where do I start here? I'm not an expert with CSS, I don't write it often and need to look up many things when I do. That said, I found the CSS work in this dashboard more frustrating than I'd hoped between the Plotly charts and the Shiny app. The complexity of working in CSS styling to Shiny depends on what you'd like to do.

Often I could get away with wrapping my web content into a div and then styling that div to view how I wanted, but sometimes when I wanted to style part of the shiny layout itself (like a nav bar or sidebar), I had to spend a lot of time in the Chrome Developer Tools window inspecting elements to isolate which element I really wanted. Shiny uses lots of calculated bootstrap CSS values and classes to style its content out of the box. Overriding or changing it can be tricky.

In at least one case I wanted to restyling parent level bootstrap attributes for a shiny object, where applying my own class to it in the python code was too abstract to style what I wanted in that object, which was nested down in its generated html code. As a result all objects of that type were styled the same way and I couldn't figure out how to change that.

If you look at my styles.css in the www folder of the dashboard, you'll see things like:

```
/* used to get rid of blue ghost border for open accordion button */
.accordion .accordion-button:not(.collapsed):focus {
    box-shadow: none;

}

/* used to get rid of blue ghost border for closed accordion button
*/
.accordion-button:focus {
    box-shadow: none;
}

.accordion-button:not(.collapsed) {
    background-color: rgb(0, 0, 0, 0);
}
```

Because of styling decisions like this, if I used an accordion again in the app it would have the same styling applied. I had to limit myself to one accordion for the app.

I'm sure as I work on more of these I'll get better at styling specific shiny components - and specific parts of shiny components if I want to, but for now I'll have to endure the learning curve.

Two style adjustments that I enjoyed working on were the rosette image on the nav bar and the head stock on the end of the waffle chart on the session tab. Fun fact, the images for these were from actual pictures of my daily driver guitar (Yamaha CG-101MS) - of course with some Gimp work.

Other Tools/Services Used

There were a number of other services that didn't directly affect the critical technology stack but were nevertheless instrumental as support tools:

- GitHub - Source code storage location
- VS Code - Used as the IDE
- Gimp - For image manipulation
- imagecompressor.com - used to prepare high res images for the web
- freepik.com - for some of the background images used in the Goals tab
- Ocrascan Validator - used to check Open Graph meta tags to see how the share link is displayed on social media

- qr.io - used to generate QR codes for the Data Entry and Dashboard apps