Contents

1 1. Klepsidra timer: First steps and prototyping 1
- 1.1 Introduction . . . 1
- 1.2 Inspiration . . . 2
- 1.3 Towards a more useful activity timer . . . 3
- 1.4 Data representation . . . 3
  - 1.4.1 Date format support . . . 4
- 1.5 Primitive implementation . . . 5
- 1.6 Choice of data store . . . 6
- 1.7 References . . . 7

2 2. Klepsidra datetime-local timestamp manipulations 7
- 2.1 Summary . . . 8
- 2.2 Using the date_time_parser library . . . 9
- 2.3 Using the Timex library . . . 9
- 2.4 Concluding thoughts . . . 11
- 2.5 References . . . 11

3 3. Primary uses: analytics, budgeting and billing 11
- 3.1 On billing . . . 11
- 3.2 Personal time tracking and analytics . . . 12
- 3.3 Budgeting and time estimation . . . 13
- 3.4 Billing . . . 14
- 3.5 Time increments used in billing . . . 15
  - 3.5.1 Compiled list of all sensible billing increments to be used 16
- 3.6 References . . . 17

1 1. Klepsidra timer: First steps and prototyping

```
Mix.install([
  {:kino, "~> 0.12.3"},
  {:kino_db, "~> 0.2.6"},
  {:exqlite, "~> 0.19.0"}
])
```

1.1 Introduction

Klepsidra is a simple task timer prototype, developed in Elixir initially deployed in a Livebook.

The purpose of this program is to time business and personal activities, for commercial purposes, analytics, data collection and better time estimation, and personal development.

1.2 Inspiration

The need to know where time goes has always been a driving force. Whether it's for personal tracking or professional billing needs, it has always been needed, and it's a need I filled with a range of historical tools. Of these, I have always found Org-Mode to be one of the most useful.

Org Mode—just like a Livebook—is intended to be written in hiererarchical sections, and just as Livebook it is a plain text file. A clock can be started in any section the author is currently working in, and all the software does is add a metadata `:LOGBOOK:` block under the relevant section. As a plain text file, the *clocking in* and *clocking out* times are appended to that block, along with calculated durations, all in a specified, easily readable (by humans), and parseable (by software) text (Hinman, 2023).

Being plain text makes it easy to work on and modify when needed—imagine all the times you've forgotten to start the clock on time, or worse, have left it running. And the beauty of all this is that a simple command can be run at any time to aggregate all the timers in a document—or even across Org Mode documents—constructing a timesheet with timing breakdowns.

So, the system is simple, malleable, and very useful. Yet it has major shortcomings—which is not a flaw, but an artefact of the system—making it problematic.

- It is Emacs software, so it isn't easily portable without *always* using Emacs
- It *is* plain text, locating the data to that file, or a carefully listed set of files; it is not portable
- The timers are limited to the purpose of that document and can be categorised according to any number of purposes, forcing the use of Org Mode, and therefore Emacs, for *absolutely everything*, which is not portable
- For every timer, a duration is calculated and stored, there is no provision for any further metadata, such as a fine-grained description of what was done, categorisation of activity through tagging, calculation of *blocks of time* spent and billable
- Timers are weakly linked to the section, and strongly linked to the document

The last point deserves a little explanation. Much of my writing and programming begins as an exploratory process. In the course of the exploration, I change section names by necessity, split sections into deeper subsections, combine smaller sections into a more encompassing one, and more. In doing that, I necessarily change the scope of the work within the section, which necessarily results in the particular attached timings becoming *untrue*. The absolute time spent is accurate, but it no longer accurately reflects the work within.

Yes, timers (being plain text) can be copied and moved around, but without any further metadata attached to individual timings, they individually lose all meaning, and this undermines their purpose to a large degree.

1.3 Towards a more useful activity timer

The conteders have been named, but have fallen short for me. What do I want?

1. A system that is quick and easy to use
2. Data which is forever portable, to any future system, process or analysis method
3. Timers which are *decoupled* from the activity and independent
4. Data format must be durable and easy to analyse, manipulate and reuse
5. Timers may be started and ended at random times due to user error and need to be easily modifiable; timing events can be missed altogether if a user isn't near any method of triggering timers, and needs to register events manually, post facto. Essentially, this must be easy to do
6. Timers which can be annotated with details of the activity measured, and categorised across multiple categories
7. An *open* system which lends itself to endless UI paradigms: web, desktop, tablet, mobile, CLI facilities at the very least, with further possibilities as necessary: timers controlled by email, text message, Telegram messenger, dedicated custom hardware (embedded devices), digitised pens and any other form of interaction deemed to reduce friction or inertiaa impeding its regular use
8. A system which provides deep and faceted insight into how time has been used, where analytics are not only possible but easy
9. A system which doesn't only record the absolute duration of activities, but records blocks of time used—according to a range of possible time accounting regimes—for the purposes of commercial billing

Beyond the clear requirements, there is a *would be nice to have* feature: locality of data. It would be desirable that the data is local to the device on which it is being used, ensuring:

- Data privacy
- High performance due to data being on the same device as the UI, the front and back end
- Low lag times, and no downtime associated with volatile network connections
- Ease of data backup

The entire concept is enshrined in an article, *Local-first software* describing the concept and its benefits at the level of detail the topic deserves (Kleppmann et al., 2019).

1.4 Data representation

Learning from Org Mode's example, only two timestamps are important: one at the instant the timer was started, and one at the end of the timing process. Together, they are one complete timed event; without both present, the timing is invalid. Besides this, any number of timing events can take place, though not

overlapping.

Records in a database table form the individual timers, each one with a start and end timestamp and additional metadata.

The pair of timestamps forms the most crucial information about each timer, and that is where the focus goes. To make portability possible as in point 2 above, and user interaction and easy modification, as in point 5, efficient representations such as UNIX time or any other "number of seconds since epoch" variant are ruled unsuitable. These are inscrutable to regular users of the sytem, and can even hamper portability if that particular variant is not trivially supported by an analytics system.

Though it may be a more verbose format, more costly to index and process, the ISO 8601 standardisation is the way to go. It is easy to read, easy to modify, it is plain text and completely portable, and well-supported across systems and data stores ('ISO 8601', 2024). Timestamps will be encoded as `YYYY-MM-DDThh:mm:ss`, where `Y` stands for year, `M` for month, `D` for date, `h`, `m`, and `s` for hour, minute and second digits, respectively, and the `T` stands in as a delimiter between date and time components.

The chosen format satisfies the most important criteria, though—as noted—at the cost of extra storage space for strings, and additional time needed for parsing and processing the string, however this is not expected to form a noticeable drag on performance until a truly large number of records exist in the database, if even then. Should this ever become a problem, there are strategies which may be employed to remedy data access and analysis.

1.4.1 Date format support

Is this format widely used and supported in practice? Elixir's various time and date libraries support it. For example, `NaiveDateTime`—chosen over `DateTime` because it is timezone agnostic, a desirable quality for a timer—gets the current time, easily converting it to an ISO 8601 string, and vice versa as shown in the following examples (NaiveDateTime — Elixir v1.16.2, 2024).

```
NaiveDateTime.local_now() |> to_string()
```

```
"2024-03-11T18:03:49" |> NaiveDateTime.from_iso8601!()
```

Presently, the only two other important tools are PostgreSQL and SQLite; how is their internal support? While there are always extensions and plugins, for development and sysadmin purposes it should be simple, not requiring further dependencies and complexity for such a simple project.

PostgreSQL provides the `timestamp without time zone`, which is an 8601 date and time representation (8.5. Date/Time Types, 2024).

SQLite does not provide a dedicated data type, but its date and time functions will store ISO 8601 strings as text, and are designed to convert between this for-

mat and representations in real and integer formats (Date And Time Functions, 2023).

This is enough of a green light to this format in terms of portability and software support, to go forward to the next stage: implementation.

1.5 Primitive implementation

To start off with, let's create a primitive version of the timer we want. Very simply, every timed activity needs to have a start timestamp—when the timer was started—an end timestamp—when it was stopped—a calculated duration in minutes (to avoid recalculation), a description of the activity timed, and a list of tags applied to the activity.

In this primitive version, all the timestamps will be stored in a simple list structure, `activity_timers`.

```
activity_timers = []
```

It is helpful to create a simple structure to store this information, ensuring a consistent and robust storage.

```
defmodule Klepsidra.ActivityTimer do
  @doc """
  An activity timer structure. Ensures that `NaiveDateTime` stamps are stored in starting
  and ending pairs, making it easy to spot _dangling_ timers.

  To avoid expensive duration recalculation, the duration integer and time unit will be
  stored for the calculated timer duration. There is a shadow pair of _reported_ duration
  and time units, used as the basis for future reporting needs. For example, some
  professionals bill in six-minute intervals, so any duration is automatically rounded up
  to the nearest six-minute multiple.
  """
  @enforce_keys [
    :start_stamp
    # :end_stamp,
    # :duration,
    # :duration_time_unit,
    # :reported_duration,
    # :reported_duration_time_unit
  ]
  defstruct start_stamp: nil,
            end_stamp: nil,
            duration: 0,
            duration_time_unit: :minute,
            reported_duration: 0,
            reported_duration_time_unit: :minute,
            description: "",
```

```
            tags: []

  @type t :: %__MODULE__{
          start_stamp: NaiveDateTime.t(),
          end_stamp: NaiveDateTime.t(),
          duration: non_neg_integer,
          reported_duration: non_neg_integer,
          description: String.t(),
          tags: List.t()
        }

  def start_new_timer(timers_list) when is_list(timers_list) do
    [%Klepsidra.ActivityTimer{start_stamp: NaiveDateTime.local_now()} | timers_list]
  end

  def stop_timer(
        [%Klepsidra.ActivityTimer{start_stamp: start_stamp, end_stamp: nil} = current_timer
          _timers_list
      ) do
    end_stamp = NaiveDateTime.local_now()
    duration = NaiveDateTime.diff(end_stamp, start_stamp, :minute) + 1

    current_timer
    |> Map.put(:end_stamp, end_stamp)
    |> Map.put(:duration, duration)
    |> Map.put(:duration_time_unit, :minute)
    |> Map.put(:reported_duration, duration)
    |> Map.put(:reported_duration_time_unit, :minute)
  end
end
```

Let's try to start a new timer:

```
activity_timers
|> Klepsidra.ActivityTimer.start_new_timer()
|> tap(fn _ -> :timer.sleep(139_000) end)
|> Klepsidra.ActivityTimer.stop_timer()
```

This little test demonstrates that the above structure is a sufficient starting point for satisying the desiderata laid out previously. The next step is to convert this into an SQLite table.

1.6 Choice of data store

Elixir's Phoenix framework uses Ecto as its object-relational data mapping layer (ORM). Ecto prefers PostgreSQL as a data store, and for many reasons, this is

an excellent choice. Klepsidra is a small and feature-limited system, at least at this point, when it is merely being built as a demonstration of its ability to fulfill specified needs. For the purposes of agile development, speed and overall ease of deployment, including the relevant consideration of a local-first application, this system will use SQLite as its data store.

Despite PostgreSQL being the preferred target database, SQLite is well-supported by Ecto, and that is the datebase which will be targeted throughout the prototype development phase, while always striving to keep compatibility with Postgres, preserving the option of future migration to that data store. What is really exciting is a new database replication system, Electric SQL, aiming to foster local-first experiences by synchronising a local-first SQLite database—with data translation—to a network- or cloud-available PostgreSQL, or even a local PostgreSQL to a cloud-hosted one (ElectricSQL - Sync for Modern Apps, n.d.).

1.7 References

- ISO 8601. (2024). In Wikipedia. https://en.wikipedia.org/w/index.php?title=ISO_8601&oldid=121106
- Hinman, L. (2023, March 20). Clocking time with Org-mode. https://writequit.org/denver-emacs/presentations/2017-04-11-time-clocking-with-org.html
- Dominik, C., & Guerry, B. (2024, March 9). Org Mode. https://orgmode.org
- Chapter 8. Data Types. (2024, February 8). PostgreSQL Documentation. https://www.postgresql.org/docs/16/datatype.html
- 8.5. Date/Time Types. (2024, February 8). PostgreSQL Documentation. https://www.postgresql.org/docs/16/datatype-datetime.html
- NaiveDateTime—Elixir v1.16.2. (2024, March 10). https://hexdocs.pm/elixir/1.16.2/NaiveDateTime.htr
- Kleppmann, M., Wiggins, A., Hardenberg, P. van, & McGranaghan, M. (2019, April 1). Local-first software: You own your data, in spite of the cloud. https://www.inkandswitch.com/local-first/
- ElectricSQL - Sync for modern apps. (n.d.). Retrieved 12 March 2024, from https://electric-sql.com/

2 2. Klepsidra datetime-local timestamp manipulations

```
Mix.install([
  {:timex, "~> 3.7"},
  {:date_time_parser, "~> 1.2"}
])
```

2.1 Summary

In the process of working on the timer in LiveView, one aspect of friction has come up: parsing HTML's `datetime-local` datetime stamp.

Essentially, JavaScript elides the seconds value, unless it has been entered or is non-zero. When the seconds are zero, only the hour and minute components are stored and transferred over the wire. This provides a better user experience, particularly with manually created timers, since it isn't desirable to ask the user an input for the seconds component. As it is an effectively meaningless precision in the context of timing activities, it's best to avoid it.

This causes data interoperability problems. As a result of the above, datetime stamps passed by the front end will therefore always be in the `YYYY-MM-DD hh:mm` format, which is *almost* ISO 8601:2019 compatible ('ISO 8601', 2024). But not quite since it misses the seconds component (HTML Standard, n.d.). Attempting to ingest that non standards-compliant string, `NaiveDateTime.from_iso8601/1` will fail with an error. This adds a friction point when ingesting data from the front end, needing the adding of a *seconds* time component.

> 2.3.5.5 Local dates and times
>
> A local date and time consists of a specific proleptic-Gregorian date, consisting of a year, a month, and a day, and a time, consisting of an hour, a minute, a second, and a fraction of a second, but expressed without a time zone. [GREGORIAN]
>
> A string is a valid local date and time string representing a date and time if it consists of the following components in the given order:
>
> A valid date string representing the date A U+0054 LATIN CAPITAL LETTER T character (T) or a U+0020 SPACE character A valid time string representing the time A string is a valid normalized local date and time string representing a date and time if it consists of the following components in the given order:
>
> A valid date string representing the date A U+0054 LATIN CAPITAL LETTER T character (T) A valid time string representing the time, expressed as the shortest possible string for the given time (e.g. omitting the seconds component entirely if the given time is zero seconds past the minute)

The reverse is also a problem, although a smaller one, when stringified `NativeDateTime` structures are passed as values into HTML forms. Since there is a seconds component in these structures, converting it to a string and applying it to HTML elements as a *value*, now produces a less readable value in the input field. Anything that further complicates fast user comprehension must be removed. In these cases, the seconds component must be stripped out.

The ideal solution is to rely on Elixir's core functionality, but in the interest of better maintainability, the simplest external libraries are a good solution.

2.2 Using the date_time_parser library

The `date_time_parser` library seems like a simple solution to the first problem, gracefully ingesting incomplete datetime strings passed from the HTML user interface, parsing them to the correct extended ISO 8601 specification.

Let us specify two example string formats to be expected, one with a `T` delimiter between the date and time components, and the other with just a space.

```
html_datetimestamp_t_delimited = "2024-03-15T15:01"
html_datetimestamp_space_delimited = "2024-03-15 15:01"
```

```
"2024-03-15 15:01"
```

Both of these will be passed into the `parse_datetime/1` function. To see both results, they are returned in a tuple.

```
{
  DateTimeParser.parse_datetime!(html_datetimestamp_t_delimited),
  DateTimeParser.parse_datetime!(html_datetimestamp_space_delimited)
}
```

```
{~N[2024-03-15 15:01:00], ~N[2024-03-15 15:01:00]}
```

This experiment satisfies the desired input parsing, returning proper `NativeDateTime` structures, with zeroed seconds components. Our core requirement of handling only these structures internally has been met, and this conversion will take place at the application boundary layer.

2.3 Using the Timex library

As the *"…richest, most comprehensive date/time library for Elixir…"*, `Timex` was considered (Getting Started — Timex v3.7.11, 2023). While undoubtedly a heavy library with even more extencive dependencies than `date_time_parser`, it really does provide a rich choice of date and time calculation and manipulation functionality.

Let's try to parse the inbound datetime-local stamp with Timex, as before. Timex' `parse/2` function takes the input string as the first argument, and a `format_string` as the second, which can be one of two types. Timex offers its own default directive format, which is simple to read and memorise, and the standard strftime format, which many people are used to.

Let's see both in action, just for comparison.

```
[
  {
```

```
    Timex.parse!(html_datetimestamp_t_delimited, "%Y-%m-%dT%H:%M", :strftime),
    Timex.parse!(html_datetimestamp_space_delimited, "%Y-%m-%d %H:%M", :strftime)
  },
  {
    Timex.parse!(html_datetimestamp_t_delimited, "{YYYY}-{M}-{D}T{h24}:{m}"),
    Timex.parse!(html_datetimestamp_space_delimited, "{YYYY}-{M}-{D} {h24}:{m}")
  }
]
```

```
[
  {~N[2024-03-15 15:01:00], ~N[2024-03-15 15:01:00]},
  {~N[2024-03-15 15:01:00], ~N[2024-03-15 15:01:00]}
]
```

Unlike `date_time_parser`, Timex provides formatting functions as well. Can it be used at the boundary to easily convert `NativeDateTime` timestamps to a string, *and* elide the seconds component?

Towards this, there is a `format/2` function, offering an identical conversion in the opposite (outbound) direction as well. Let's see it in action.

```
datetime_stamp = ~N[2024-03-15 15:01:00]
```

```
~N[2024-03-15 15:01:00]
```

```
[
  {
    Timex.format!(datetime_stamp, "%Y-%m-%dT%H:%M", :strftime),
    Timex.format!(datetime_stamp, "%Y-%m-%d %H:%M", :strftime)
  },
  {
    Timex.format!(datetime_stamp, "{YYYY}-{M}-{D}T{h24}:{m}"),
    Timex.format!(datetime_stamp, "{YYYY}-{M}-{D} {h24}:{m}")
  }
]
```

```
[{"2024-03-15T15:01", "2024-03-15 15:01"}, {"2024-3-15T15:01", "2024-3-15 15:01"}]
```

```
13:38:48.058 [debug] Tzdata polling for update.

13:38:49.044 [debug] Tzdata polling shows the loaded tz database is up to date.

18:03:28.850 [debug] Tzdata polling for update.

18:03:30.631 [debug] Tzdata polling shows the loaded tz database is up to date.

02:13:35.209 [debug] Tzdata polling for update.
```

```
02:13:36.252 [debug] Tzdata polling shows the loaded tz database is up to date.

03:36:28.253 [debug] Tzdata polling for update.

03:36:30.007 [debug] Tzdata polling shows the loaded tz database is up to date.

15:30:43.429 [debug] Tzdata polling for update.

15:30:44.573 [debug] Tzdata polling shows the loaded tz database is up to date.
```

And it returns properly formatted `T` and space-delimited datetime strings, with the seconds component elided, ready to pass to HTML elements' value slots.

This satisfies both directions of boundary conversions needed, and demonstrates the best path to follow.

2.4 Concluding thoughts

Given that it works almost as easily, while providing a wealth of functionality to be used in future development, such as filtering timers to reveal only those in a time period, it is more sensible to use the Timex library for boundary conversions.

2.5 References

- HTML Standard. (n.d.). Retrieved 15 March 2024, from https://html.spec.whatwg.org/multipage/comm microsyntaxes.html#valid-normalised-local-date-and-time-string
- ISO 8601. (2024). In Wikipedia. https://en.wikipedia.org/w/index.php?title=ISO_8601&oldid=121106
- DateTimeParser—DateTimeParser v1.2.0. (2023, December 6). https://hexdocs.pm/date_time_parser/DateTimeParser.htm
- Getting Started—Timex v3.7.11. (2023, May 4). https://hexdocs.pm/timex/getting-started.html

3 3. Primary uses: analytics, budgeting and billing

3.1 On billing

Klepsidra is a mixed-purpose product: it needs to be a platform for analytics on how and where time is spent, on a personal and commercial level. It aspires towards the kind—without the personally-invasive level of detail—of tracking that Stephen Wolfram's method achieves. An important aspect of this is that it manages client billing, making it painless, quick and easy to use.

Essentially, time tracking is useful if it helps to:

- Bill accurately for your time
- Track productivity and see if tasks are taking longer than they should
- Discover and prioritize high-value tasks to boost business profits
- Discover and reduce non-billable tasks to minimize *wasted* time
- Understand your resource availability by estimating how many hours will be needed to complete tasks
- Schedule resources accurately
- Pay employees fairly for time spent on time-intensive tasks

The system will be considered successful if it manages all of the above.

3.2 Personal time tracking and analytics

Most of the day cannot be spent on work. Practitioners of every field must fill their time by maintaining their existing knowledge, putting into practice—even if bona fide—their skills to keep sharp, and learning and developing new skills and abilities. They will spend time branching out from their area completely, learing marketing, advertising, accounting, design, speaking a new language, and other areas but their own. It's important to do this at a sustained pace, and consistency may be more important than the degree of measurable progress. It would be desirable to track these forays, the amount of time invested into learning or practising, a note describing what was done and learned in these sessions, and also a snapshot of consistency.

Insight into this will pay dividends in the future. Not only will it be easy to see how much time has been invested into any area, but it will be glaringly obvious if too much time has been spent, and if it may be better to [re]focust future attentions. The mind easily forgets things, especially without consistency. Data will bear out how consistent efforts have been, and will help foster regular, daily practice; by presenting time invested on a heatmap—not dissimilar to the way GitHub records our commitments to code—it is my hope that this will instigate a more serious approach towards learning and experimenting in me, for example.

A personal failing I have witnessed in me is that, while in the midst of it, I am sure what I have done, learned and achieved; at the end of the year though, I have forgotten most of what *it* was. By recording simple but deliberated notes for each timed activity, a timeline of exactly what I have learned will miraculously appear. To some, this will help at the next review meeting, or in future pay negotiations, to others it will be the embodiment of a living portfolio.

For personal analytics, activity tracking and analytics, I need the system to:

- Show me a list of high-level activities (catrgorisations) I engage in
- Sum the time I spend on activities, in a month, quarter and year
- Display a heat map, by day, of activities engaged in, by time spent
- Build a timeline and report, reminding me of things I have learned

- Calculate proportions between time spent on personal pursuits versus commercial ones
- Record activity times, not only durations, as it is valuable to know *when* I engage in certain activities, informing future personal time management

The system must differentiate from billable time, as well as other commercial time, e.g. time used internally for administrative duties, accounting, marketing and any other activities which are neither billable to a client, nor reflective of personal and professional development.

3.3 Budgeting and time estimation

One of the greatest problems I face is being able to estimate time needed for a project or task, or to quote a flat fee, while achieving any kind of accuracy. It's not a unique problem and it isn't even limited to the technical sector, though it's most noticeable there. Many engineering projects—particularly civil engineering—fall prey to the same problem. Why is this? One obvious reason that's common across many industries in the desire to please clients, or at least, not to scare them away with a high cost. This tendency, coupled with tender processes, often favours the least expensive provider, to put it reductively. Without ascribing any greed or malice, this tends to lead to projects which end up egregiously over budget and time estimates.

We get into this state of affairs not merely by misrepresenting the true picture to clients; we wilfully lie to ourselves, convinced that an optimistic time and cost estimate is going to be correct. Without oversimplifying matters, this is true for at least three reasons: (1) we believe we've learned lessons from the past and we will be faster now; (2) we ignore many of the things that may go wrong; (3) we budget way too little for *unknown unknowns*, which will become massive time sinks over the course of the project.

One factor that underlies all of this is lack of information. Since in software, construction, engineering and similar fields, past projects rarely perfectly mimic future ones, using past experience is never the most accurate predictor. However, this is squarely due to a lack of data. If we had detailed records of clearly defined stages—down to very fine levels of detail—which honestly included timing not only of the final success, but also all the dead-ends, bottlenecks and problems encountered, a sufficiently large corpus of this date would become statistically significant. Mining this data will yield a probability distribution as well as the ability to plot a trendline, with known margins of error, based on past execution. The more past data we build up, the more accurate trendlines will result, with tighter margins of error.

Both of these tools would lead us to more accurate estimation in the future. It is a truism that with more data—more projects carried out—our trendline would become more accurate and meaningful. The only solution is to diligently record *all* our projects and tasks to build up our data, helping us deduce more valuable timelines from that.

To achieve this we need to record:

- All types of projects, tasks and activities
- Record time taken *honestly*, without removing or reducing time spent on unpalatable problems
- Record activities *consistently*, recording a high level of detail in our categorisation to be able to use faceted search and filtering when analysing past activities

Our activity timer must *not* allow easy modification of the actual time recorded, like an accounting system, paving the way to auditing our past successes and mistakes.

3.4 Billing

Billing is an important area, as many professions need to bill for their time, according to how much time has been used for a client. It is easy to build a primitive timer, and there are many in existence; Klepsidra has lofty goals which need sophisticated handling.

In timing activities, we strive for an *append-only* recording strategy, in a practical sense. This means that Klepsidra will not do anything to help changing the past and rewriting activity timings. However, there are many ways this can backfire.

- We forget to start a timer when we start a new task
- We forget to stop a timer on time
- We are not near any user interface exposed by Klepsidra, or there is a technology failure preventing us from accurately timing an activity
- We are unaware at the time of starting something that this is actually a timeable activity

There are surely more reasons than that, but what is important is that there will be lapses, which must be handled to result in the quality of data we desire. A design decision is to provide three types of time tracker: (1) automated tracker activated by *start* and *stop* actions; (2) a manual timer where all data available in the first timer may be manually recorded; and (3) a simple duration log, where times are not available at all, and only the task duration is recorded.

The automated timer interface will actively prevent any changing of start and end timestamps. To handle recording errors identified, an adjustment field will be provided, paired with a categorisation of correction and a description field to explain the particular circumstance. The adjustment time unit will match the actual duration unit, e.g. if the duration is counted in minutes, the adjustment will be in minutes.

The manual timer needs to be available to retroactively record activities which wasn't possible with the automated timer, either because of technical problems, use in a strict offline environment, because of a user's recording by analogue

means, and other similar reasons. In this case the interface will record all the same information, but start and end dates will be manually entered. As this is a manual entry, no adjustments will be made to the duration, since it is expected that any lost time has been accounted for in the timestamps.

The third type of recording will have only the duration field. This is believed to be necessary as there will be many practitioners and activities where accurate time recording isn't practical. Instead, they will report the actual time taken, in minutes or hours. This feature necessitates using a floating point recording of duration, particularly when hours or days are used as the recording unit.

In all three cases, a further adjusting entry is desirable: billable time. Despite being able to configure Klepsidra to round up time durations for billing, or round it down, there will be many situations in which a time increment has only just been broached, and it may be desirable to adjust the billing down to the next lowest block. Just as with duration adjustments, an adjustment category and reason should be provided. This reason may be passed through to the invoice for building up customer trust in the billing process, providing transparency. The correction will be denominated in the same time unit as the calculated duration. The billable increment (see below) is calculated from the total duration, including the billing adjustment.

Timed activities need to be categorised as billable, which will impact client invoices, personal/professional development—for personal analytics—and other for activities that don't follow either of these patterns.

3.5 Time increments used in billing

The easiest form of billing is per-hour billing. This has its own disadvantages as it is a very coarse measure, which is a contentious tug-of-war between clients who obviously prefer much finer increments, in some examples demanding down to per-minute detailed charging, and service providers who tend towards more coarse time units. There is much to be said for both tendencies, but what is really important is that there is no standard and many *billing increments* are used in commerce.

To name a few examples, those in the legal profession often use tenths of an hour—six minute increments; customer support may use the even finer five-minute increments; Medicare practitioners use eight-minute blocks; quarter-hour (15 minute) increments are used across most service providers; designers, developers and other professionals use half-hour, hour, and even two-hour blocks. In fact, this is a business-specific decision, and there are often complex policies using a combination of several of the above.

To satisfy all the above needs, Klepsidra needs to provide all the above increments for flexibility across all practical service providers' requirements. The `ex_cldr_unit` library makes it relatively easy to create custom units, based on those already in the library. The catch is that this is not a dynamic pro-

cess, rather a pre-defined and compiled one "…units are compiled into code" (Elixir-Cldr/Cldr_units, 2017/2023). This forces the hard-coding of all possible increments into code to be able to rely on all these complex time conversions, whereupon the user interface will have to be used to limit the list to only desired increments. A good explanation states "Common billing increments range from minutes to hours, each serving specific purposes and preferences within different industries and professions" (Forecast, 2024).

3.5.1 Compiled list of all sensible billing increments to be used

Below is a list of billing increments in use to some extent across a range of service providers. Is is important to keep in mind that, whichever billing increment, Klepsidra will—by default—round up towards a multiple of that increment. For example, if 13 minutes have elapsed, this would be equivalent to three 6-minute, two 10-minute, one 1-hour increments, for example, and would be invoiced pro-rata according to that calculation. Please note that, when all the increments are taken into account combinatorically, there is a wealth of billing choices here!

Billing increment	In minutes	In hours	Description
5 minutes	5	0.0833	A particularly fine increment for high precision with many predictably short tasks
6 minutes	6	0.1	Offers precision in time tracking and billing
10 minutes	10	0.1667	At six blocks per hour, it is fairly fine though an unruly fraction for billing
12 minutes	12	0.2	Balanced granularity for short-term engagements
15 minutes	15	0.25	Ideal for meetings, consultations, and tasks
18 minutes	18	0.3	Commonly used for technical tasks and reviews
20 minutes	20	0.3333	Offers balance between quarter and half-hour increments
24 minutes	24	0.4	Provides flexibility for diverse tasks
30 minutes	30	0.5	Convenient increment for routine activities
36 minutes	36	0.6	Suitable for medium-length consultations
45 minutes	45	0.75	Used for sessions requiring extended time
1 hour	60	1	Standard billing unit for project work (common in agencies)
1.5 hours	90	1.5	Provides flexibility for longer engagements

Billing increment	In minutes	In hours	Description
2 hours	120	2	Ideal for specialized tasks and consultations

That's fourteen different billing increments there, in addition to the base unit—minute—which should provide sufficient flexibility to most service providers. This is not a complex system, and precludes use by some regulated professionals such as the previously mentioned physical therapists and related providers on Medicare, for example. Successfully providing for their needs requires a more complex calculation, including time ranges, providing the ability to define staggered, unequal billing increments, as well as those where there is an initial flat fee increment, followed by finer increments once a predefined lenght of time has elapsed.

3.6 References

- Cldr.Unit—Cldr Units v3.16.4. (2023, November 2). https://hexdocs.pm/ex_cldr_units/Cldr.Unit.html
- Elixir-cldr/cldr_units. (2023). [Elixir]. Elixir CLDR. https://github.com/elixir-cldr/cldr_units (Original work published 2017)
- Forecast. (2024, March 18). What are Billing Increments? https://www.forecast.app/learn/what-are-billing-increments