…Expand for EKG code

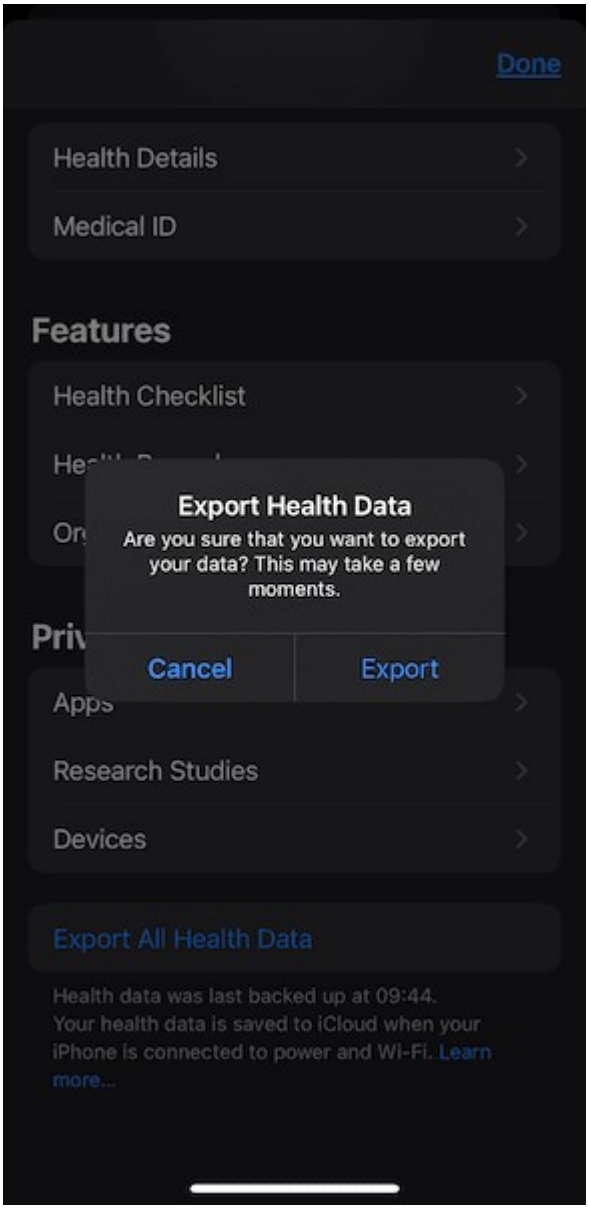
```
library(hrbrthemes)
library(elementalist) # remotes::install_github("
teunbrand/elementalist")
library(ggplot2)

read_csv(
  file = "~/Data/apple_health_export/electrocardiograms/ecg_2020-
09-24.csv", # this is extracted below
  skip = 12,
  col_names = "µV"
) %>%
  mutate(
    idx = 1:n()
  ) -> ekg

ggplot() +
  geom_line_theme(
    data = ekg %>% tail(3000) %>% head(2500),
    aes(idx, µV),
    size = 0.125, color = "#cb181d"
  ) +
  labs(x = NULL, y = NULL) +
  theme_ipsum_inter(grid="") +
  theme(
    panel.background = element_rect(color = NA, fill = "#141414"),
    plot.background = element_rect(color = NA, fill = "#141414")
  ) +
  theme(
    axis.text.x = element_blank(),
    axis.text.y = element_blank(),
    elementalist.geom_line= element_line_glow()
  )
```

Apple Watch owners have the ability to export their tracked data and do whatever they like with it. Since it's Valentine's Day, I thought it might be fun to show two ways to read heart rate data from these exports.

Why *two* ways? Well, I've owned an Apple Watch off-and-on ever since the first generation device, and when Apple says you can export *all* your data, they mean *all*. The `apple_health_export.zip` archive is generated by going to the "Health" iOS app, tapping your avatar in the upper left, then scrolling down and tapping the export button:



(NOTE: I suggest saving it to and then downloading it from iCloud vs using local AirDrop to your system.)

This compressed file is a deceivingly ~58 MB in size. Opening it up results in a directory tree of nearly 3 GB of consumed drive space O_o. That tree has the following structure:

```
fs::dir_tree("~/Data/apple_health_export", recurse = 1)
## ~/Data/apple_health_export
## ├── electrocardiograms
## │   └── ecg_2020-09-24.csv            # 122 KB
## ├── export.xml                        # 882 MB
## ├── export_cda.xml                    # 950 MB
## └── workout-routes                    #  81 MB
##     ├── ...
##     ├── route_2021-01-28_5.21pm.gpx
##     ├── route_2021-01-31_4.28pm.gpx
##     ├── route_2021-02-02_1.26pm.gpx
##     ├── route_2021-02-04_3.52pm.gpx
##     ├── route_2021-02-06_2.24pm.gpx
##     └── route_2021-02-10_4.54pm.gpx
```

The heart rate data is in the just-under 1 GB `export.xml` and is mixed in with all the other data points Apple records. They look like this:

Note that newer records of this type are not empty tags.

While dealing with gigabyte+ XML files are not nearly as untenable as they used to be in R, building a parsed XML tree in memory for all of those records will take up a non-insignificant amount of RAM (we'll see how much below). Since I want to start playing with this data more often I decided to try two approaches: one that processes the XML in streaming "chunks" and one that does it the way you're likely used to (if you're unfortunate enough to have to work with XML regularly).

Streaming 💙 Beats

We'll start with the streaming approach, which means using the venerable {XML} package, which has `xmlEventParse()` which is an *event-driven* or SAX (Simple API for XML) style parser which process XML without building the tree but rather identifies tokens in the stream of characters and passes them to handlers which can make sense of them in context. Since we're going old-school, we'll also use {data.table} to get a tidy dataset to work with.

We're going to be finding heart rate records and storing the data from them into a list, so we'll need to make room for them and use indexed-based value assignments to avoid making thousands of copies with `append()`. To figure out how much room we'll need I'm going to "cheat" a bit and use ripgrep to count how many `HKQuantityTypeIdentifierHeartRate` records exist and use that result to reserve list space:

```
library(XML)
library(data.table)

nl <- system("rg -c 'type=\"HKQuantityTypeIdentifierHeartRate' ~/Data
/apple_health_export/export.xml", intern = TRUE)
records <- vector(mode = "list", as.numeric(nl))
idx <- 1
```

There are just under 790K records buried in that file. The `xmlEventParse()` function has a `handlers` parameter which takes a list named functions for various events. The event we care about is the one where we start processing an XML element, which is unsurprisingly called `startElement`. In it, we'll only process `HKQuantityTypeIdentifierHeartRate` records and further only care about data since 2019:

```
invisible(xmlEventParse(
  file = "~/Data/apple_health_export/export.xml",
  handlers = list(

    # process at element start

    startElement = function(name, attrs) {

      # only care about the heart rate recs

      if ((name == "Record") && (attrs["type"] == "
```

```
HKQuantityTypeIdentifierHeartRate")) {

        # only care about records >= the year 2019

        if (substr(attrs["endDate"], 1, 4) >= 2019) {

          # if we find them, add them to the list (note the <<-)
          records[idx] <<- list(as.list(unname(attrs[c("endDate",
"value")]))) # not using names reduces memory
          idx <<- idx + 1

        }
      }
    }
  )
))
```

At this point we have a list of all those records and have taken the R session memory from 131 MiB to 629 MiB (so, we're eating about ~500 MiB of RAM with that call), and it took around 34 painful seconds to process the XML file.

Now, we'll use {data.table} to tidy it up:

```
records <- records[lengths(records) != 0]         # get rid of any list
elements we didn't use

records <- rbindlist(records, use.names = FALSE)  # make a data frame
setattr(records, 'names', c("ts", "rate"))

records[, c("ts", "rate") := list(
  as.POSIXct(ts, format = "%Y-%m-%d %H:%M:%S %z"),
  as.integer(rate)
)]
##                          ts rate
##      1: 2019-02-12 15:19:54   69
##      2: 2019-02-12 15:26:11   90
##      3: 2019-02-12 15:31:33   92
##      4: 2019-02-12 15:34:24   89
##      5: 2019-02-12 15:57:33  120
##     ---
## 734526: 2021-02-13 10:17:08  118
## 734527: 2021-02-13 10:26:50  124
## 734528: 2021-02-13 10:22:56  110
## 734529: 2021-02-13 10:34:56   98
## 734530: 2021-02-13 10:39:34   99
```

That took around 4.5 seconds, and when the R garbage collector kicks in we're now consuming ~695 MiB, so not much more than the previous step.

So, ~38s for the ingestion & conversion, and a maximum of ~695 MiB in play at any time during the R session. Let's see how the new/modern way (i.e. {xml2}) compares.

Modern 💙

Unless I missed something in the {xml2} index page, there is no equivalent streaming processor, so we have to read the entire document into active RAM:

```
library(xml2)
library(tidyverse)

records <- xml2::read_xml("~/Data/apple_health_export/export.xml")
```

This operation takes 15.7s and the R session now consumes ~5.8 GiB of RAM. That is a “G”, as in gigabyte.

Now, we’ll find all the records that we care about (as above). We’ll do this via a modest XPath selector:

```
xml_find_all(
  records,
  xpath = "
    .//Record[
         @type = 'HKQuantityTypeIdentifierHeartRate' and
         (starts-with(@endDate, '2019') or
          starts-with(@endDate, '2020') or
          starts-with(@endDate, '2021'))
      ]"
) -> records
```

That operation took around ~6.5s and we’re still consuming around 6.23 GiB of RAM.

Now, we’ll tidy that up:

```
tibble(
  ts = records %>%
    xml_attr("endDate") %>%
    as.POSIXct(format = "%Y-%m-%d %H:%M:%S %z"),
  rate = records %>%
    xml_attr("value") %>%
    as.integer()
) -> records

records
## # A tibble: 734,530 x 2
##    ts                   rate
##
##  1 2019-02-12 15:19:54    69
##  2 2019-02-12 15:26:11    90
##  3 2019-02-12 15:31:33    92
##  4 2019-02-12 15:34:24    89
##  5 2019-02-12 15:57:33   120
##  6 2019-02-12 15:44:09    80
##  7 2019-02-12 16:03:24   110
##  8 2019-02-12 16:13:08   118
##  9 2019-02-12 16:08:10   100
## 10 2019-02-12 16:15:04    95
## # … with 734,520 more rows
```

That took around 10.4s and, after garbage collection happens, we're back to a much more reasonable ~890 MiB of consumed RAM after a workflow maximum of over 6 GiB, taking a total of ~32.6 seconds.

FIN 💙

If/when memory is tight, it's nice to have some alternatives besides "get a bigger box", and this is one approach (there are others) for performing this type of XML surgery in R.

Stay safe/strong, folks.