Inclusion Dashboard Handover Guide

Mark Burdon, 22nd March 2023

Table of Contents

[A. Project timelines, deliverables, etc. 1](#_Toc130464609)

[B. RShiny app structure and functioning 3](#_Toc130464610)

[C. Providing a data download 5](#_Toc130464611)

[D. Local and national summary graphics 5](#_Toc130464612)

[E. Naming conventions 6](#_Toc130464613)

# Project timelines, deliverables, etc.

Information on the PMO aspects of the project can be found in the submission to ministers (currently finishing being drafted. Slightly older but more detailed documents can be found here:

[Local Accountability and Improvement Division - Work planning - All Documents (sharepoint.com)](https://educationgovuk.sharepoint.com/sites/LocalAccountabilityandImprovementDivision/Shared%20Documents/Forms/AllItems.aspx?FolderCTID=0x012000BEC72871FD13434ABA1E3D37F16232BC&id=%2Fsites%2FLocalAccountabilityandImprovementDivision%2FShared%20Documents%2FPerformance%20and%20Accountability%2F2%2E%20Data%20and%20transparency%2FInclusion%20dashboard%2FWork%20planning&viewid=d3fdd596%2D4761%2D4bde%2D93bd%2Df19940a52410)

In brief, once complete, the dashboard will be made “semi-public”: its code will be on Github and it will be hosted on shinyapps.io, but behind a user authentication system.

To progress to this stage, it needs approval that it’s compliant with DfE’s dashboard standards.

An email I got from Charlotte summarises this: “Here is the guidance for the steps/procedures to build a publishable dashboard: [Best practice for publishing dashboards](https://rsconnect/rsc/stats-production-guidance/dashboards.html#Dashboard_procedure_checklist)

An overview of the main things would be:

* Build in [UI/unit tests](https://rsconnect/rsc/stats-production-guidance/dashboards.html#ui-tests) from the beginning,
* Add and regularly run the [tidy code function](https://github.com/dfe-analytical-services/shiny-template/blob/7e560e68885ddb83c8077c0989e2b4232f013bc8/global.R#L43) from the beginning (keeps your code in a tidy format),
* Keep the [accessibility testing](https://rsconnect/rsc/stats-production-guidance/dashboards.html#Accessibility_testing) in mind. We usually do this towards the end but depending on your time and resource, it could be useful and save time in the long run if you check it as you go!
* Keep [performance testing](https://rsconnect/rsc/stats-production-guidance/dashboards.html#performance-testing) in mind (i.e. the speed of your dashboard, is anything slow to load etc).”

The process therefore is:

1. Completing the initial dashboard development such that is it a fully working product, the data has been QA’d, and you are happy to share it with local authority partners and other users for testing. Use the tidy\_code() function to ensure the code meets the tidy standards.
2. Ensuring that UI and unit testing has been implemented using {shinytest}. **This is mostly implemented, but some types of error code cause the testing to stop, and at the time of writing some of the later tests are erroring out.** I recommend enlisting the support of the Statistics Development Team (Cam, Rich and Charlotte) for help to iron out where the last errors are preventing the tests from running. At the very least, the EHCP timeliness benchmarking chart is one of the culprits.
3. [Accessibility testing](https://rsconnect/rsc/stats-production-guidance/dashboards.html#Accessibility_testing) – I haven’t done this yet, though the plotly graphs’ colours should all pass.
4. [Performance testing](https://rsconnect/rsc/stats-production-guidance/dashboards.html#performance-testing) – Charlotte previously did this using profvis and it was acceptable at the time, but a review would be needed before publication.
5. Submit the dashboard for final approval to Tanya McCormack (or David Simpson) as DD for publication.
6. Submit the dashboard to the Statistics Development Team for approval. When approved they will help transfer the dashboard over to shinyapps.io and github.

# RShiny app structure and functioning

The below diagram shows the main scripts and data files that generate the dashboard. More detail is provided below.

Diagram

Description automatically generated

Conceptually the process of the dashboard displaying starts with the **data\_preparation.R** script, which is in the /prep/ folder and which also sources the summary\_prep.R script. These scripts import the source data from the /data/ folder; URLs are provided for where the data came from. The data is divided into folders, and where possible (i.e. for DfE data) the original data-guidance.txt file is provided.

The script then performs an initial wave of processing on the raw data. To speed up the dashboard displaying, this preparation only needs to be run when the data is changed, and it saves down **prepared\_data.Rdata**, which contains all the post-processing data objects. It also performs the analysis of which LAs overlap with which (former) CCGs and creates an LA to CCG lookup table, such that when you click on Lancashire LA in the LA menu you can only select a CCG that has some overlap with Lancashire’s boundaries.

When the dashboard app starts, the **global.R** script runs. This script loads in the R packages used in the app, loads in the prepared\_data.Rdata file, defines styling defaults for fonts, colours and graphs, and defines the lists of LAs and CCGs for drop-down menus. It also sources the **functions.R** script, which makes a set of custom functions available to the app.

The **server.R** script defines the interactive objects available to the **ui.R** (the user interface). This essentially is performing the last steps of data processing needed to display what the user wants, but unlike the previous scripts server.R has access to what the user has selected. The local authority selected by the user is a character string saved as input$la\_choice, for example.

The code creating these objects is only triggered when needed – hence the desire to do as much data processing as possible in advance, and to perform app-wide functions in the global.R script (so they are done once, and ideally not when the user loads the dashboard). When the user changes their choice of LA, input$la\_choice updates and the objects that use that variable as a filter will in turn update what they are sending to the UI. Note that the order and position of the objects in the script does not affect their placement. This just creates a set of objects to be called on when neeed.

Within server.R, several types of object are created:

1. **Dynamic UI elements:** The first is dynamic UI elements using either renderUI() to create a UI object (in this case a region drop-down menu) from scratch, or updateTabsetPanel(), which is effectively a hyperlink to a particular tab. Note that objects that are created start with **output$**, e.g. **output$region\_choice\_out**.
2. **Plotly graphs:** The second type of objects is Plotly objects created with renderPlotly();These must all have unique names e.g. output$ks2\_attainment\_la\_time. See the section “Naming conventions” for more details on the names of these objects. Within the ({}), a Plotly object is created that is then passed on to the UI; any dataframes or other variables that are set within the brackets is temporary and won’t persist into other objects – and that principle applies to all the output$ objects.

Plotly graphs have some attractive visual qualities but often have quirks that are difficult to remedy as the R documentation is very poor. I have created some functions to carry out common changes e.g. making the legend horizontal, formatting the England average line, removing the button bar, etc.

1. **Girafe graph:** The third type is only created once, and that is the LA summary graph created with {ggiraph} and renderGirafe(). This is a Plotly alternative that sticks more closely to ggplot2 conventions. If I was starting afresh I would consider using it across the board. Note that ggiraph is a bit sensitive about font choice – it works as currently specified, but if you were to specify a particular font, it would probably display in a strange serif.
2. **Value boxes:** The Ofsted ratings and Mental Health numbers are in value boxes created with renderValueBox(). If you want to do more with value boxes, Rich Bielby has previously created code that’s part of more recent versions of the dashboard template that expands their functionality.
3. **DataTable (DT)**: Not to be confused with data.table; this is a nice HTML table object used for Ofsted data, with page buttons and row shading. It comes from the DT package using renderDataTable() and is very customisable.

The **ui.R** script sets up the user interface, though most of what you see is dictated by **dashboard\_panels.R**. Within this script, the objects from server.R are pulled through using code such as:

withSpinner(type = spinner\_type,

ui\_element = plotlyOutput("timeliness\_la\_bench", height = "110%"))))),

The withSpinner wrapper adds a “loading” spinner graphic. The text string given to plotlyOutput must be the same as the name of the object specified in server.R (after the $).

The script also contains most of the drop-down menus: these are either pickerInput() or selectInput() – both are similar but pickerInput has more options so this is preferable for the main LA selector.

A few features of note:

* The tabs (“Change over time”, “Local area comparison”) are specified by tabPanel(),
* Links are created using tags$a(href = “URL here”, “Link text here”)
* Most things are held within a box() nested in a column() within a fluidRow() – take care not to delete or add brackets or risk breaking the UI
* The script contains separate objects for the LA dashboard (“dashboard\_panel”) and England/regional dashboard (“engreg\_dashboard”).

Whenever the dashboard is uploaded to shinyapps.io, the **ui\_tests.R** script will be triggered. This shinytest-based script takes snapshots of outputs under a variety of conditions (e.g. selecting different LAs and tabs) and compares the snapshots, alerting you if the output has changed. The Statistics Development Team can share more info on this. As it stands, the script will error out because of a bad character in the output – effectively this will be the result of some kind of error text. Once the errors are resolved, the script should run smoothly.

# Local and national summary graphics

The **local summary graphic** is mostly generated as-and-when needed, to reduce processing time for the dashboard user. The code is in the summary\_prep.R script, which is sourced within the data\_preparation.R script (these could be consolidated, but the former was getting quite long and unwieldy, especially for editing).

As it is run on its own, the functions it uses are within the script itself.

The script filters each dataframe from the prepared\_data.Rdata to the most recent year of data, and filters to EHC plans only. Where it differs is that it doesn’t need to highlight a particular LA as in the benchmarking graphs, or format the output – it just calculates rankings and then combines all the metrics into one table.

I should note here that I made an executive decision to pre-calculate the metrics and focus on EHC plans for simplicity. The alternative would be to put the code into one big interactive object in the server.R file, and change the EHCP filters to be linked to a selectInput() menu; however be aware that some metrics are split All SEN/EHCP/SEN support, some don’t have an All SEN option, and destinations metrics don’t split by provision type.

The graphic is displayed by ggiraph, not plotly. Text justification in plotly is tricky, whereas ggiraph essentially just takes a ggplot object and gives it some interactive elements, so it allows for use of ggrepel’s geom\_text\_repel options.

Key features to be aware of:

* the “Highest/Median/Lowest” labels are contained in a separate dataframe created within the server.R object called text\_df. Lines are created with geom\_hline\_interactive, which is ggiraph’s way of creating a horizontal line with optional mouseover text.
* The code adds an “x” variable to the summary\_metrics dataframe with the aim of offsetting points to the side if they risk overlapping. This also gives granular control of where the labels go on horizontal lines.

The **England summary** is incomplete (my apologies). The dataframes it uses are generated in data\_preparation.R, primarily using the national\_only() function.

It has its own section of code towards the end of server.R. In each case, these code chunks each generate a dynamic UI object (the numbers at the left of each summary) and then a sparkline graph.

Much of this process uses functions (create\_box(), create\_sparkline() and format\_sparkline()). Because of differences in the underlying data, the initial processing to pull out the latest value, previous value and calculate change is done outside the functions, which then apply consistent formatting and generate the data summary and sparkline.

The presentation of the summary is finicky and uses a lot of html/css code to set text heights, and you will find that if you try to make the text side of the summary any wider, it causes the sparklines to move down underneath them. So there is little scope for changes in presentation here.

To add more metrics, replicate the existing code but feed in the relevant eng\_ dataframes and apply the filters used in server.R.

At the moment I have disabled any hover-over information in the settings for format\_sparkline(), since I couldn’t find how to remove them only for the white area plot (which was always showing the oldest data regardless of where the cursor was).

Ideally the summary graphic would have working links to the relevant parts of the dashboard.

# Object naming conventions

With one or two exceptions, the objects created by server.R follow a common pattern:

[Data item name]\_[level]\_[view]

[level] is either LA, CCG, reg (region) or nat (national), but most of them are LA or region.

So when the ks1\_phonics dataframe is brought in from prepared\_data.R, it is then turned into the following objects:

ks1\_phonics\_la\_time : a graph of KS1 phonics performance over time for the selected LA

ks1\_phonics\_la\_bench : a benchmarking graph of the latest KS1 phonics data for all LAs

ks1\_phonics\_reg\_time: a graph showing England or a selected region’s performance over time

ks1\_phonics\_reg\_bench : a graph benchmarking KS1 phonics data for all regions

There are some exceptions for e.g. destinations data that provide a third way to cut the data (select a destination, and see a line for each SEN/LLDD group).

Menus that filter down to a specific type of SEN provision (EHCP/SEN support, usually) are named after the data item, and sometimes the level e.g. ks1\_phonics\_la\_filter.

# App deployment and debugging

I recommend being familiar with at least the “Key concepts” section of the [RSConnect guide](https://dfe-r-community.github.io/posts/2022-10-27-rsconnect-guide/). Everything is set up and works fine, but familiarity with this will help you when it comes to sharing links to the dashboard.

The app’s release pipeline is here: [Inclusion DB v3 pipeline - Pipelines (visualstudio.com)](https://dfe-gov-uk.visualstudio.com/SEND%20Analysis/_releaseDefinition?definitionId=6&_a=environments-editor-preview)

Whenever you push the main branch to DevOps, it automatically deploys the app to the pre-production (pre-prod) RSConnect server, and if successful it will appear here: [Dashboard (pre-prod)](https://rsconnect-pp/rsc/content/1ea3ef3c-0233-49ac-9317-29d8cfb69688/).

Graphical user interface, application

Description automatically generated

The above image shows the pre-deployment settings for the Production server, which will update the app at [Dashboard (prod)](https://rsconnect/rsc/content/8d936632-4898-441a-a2fd-a532c4b669a7/). Essentially, it means that either I (Mark) or Stuart can approve a release (I should be removed once I go to my new role, but I may have some last minute updates to push before then). Only once I have confirmed the app is stable and nothing weird has happened to the data do I push from pre-prod to prod.

This is done by going to the Releases view and clicking on “Prod” next to the release. The screenshot below shows the Releases view as it is when a release is being prepared for deployment on Pre-Prod.

Graphical user interface, application

Description automatically generated

**Top tips:** Other than some now-fixed server issues in January and February, the #1 reason for a release failing in my experience is that I installed a new package but didn’t update renv. Always use renv::snapshot() once a new package is installed. Be cautious with updating packages – you may have access to a newer version of a package on your machine that the CRAN mirror used by the server doesn’t have access to.

# Future development

Requests for changes, feedback, and notes of ideas for what do do to improve/fix the dashboard have been collated in this spreadsheet: [Dashboard Development to-do list.xlsx](https://educationgovuk.sharepoint.com/:x:/r/sites/LocalAccountabilityandImprovementDivision/Shared%20Documents/Performance%20and%20Accountability/2.%20Data%20and%20transparency/Inclusion%20dashboard/Work%20planning/Dashboard%20Development%20to-do%20list.xlsx?d=w5707017a9d2240ac897bec7ed7071e6a&csf=1&web=1&e=Q3M9iP)

It is worth reviewing this with Sophie and Tanya to ensure we are taking account of everything agreed in the submission process.

**Data download functionality:** The original intention was to have a “Download Data” button, especially for people who need screen readers, for whom graphs are not helpful. However, this would need to either be its own product (if combining the data into one table) or essentially a dump of lots of different CSVs from EES. I discussed this with Charlotte and the solution we agreed would be best was to have a tab that describes and links to the data sources. (The data sources are on the dashboard itself, but this would be a bit more detailed for people needing to recreate the metrics). Not fully essential but this would boost the accessibility of the dashboard somewhat.

**EES API:** Ideally, when there is an API complete for EES, the dashboard will be able to more or less update automatically when there is a relevant new release from a publication. We’re still some way off that being possible.

# Assorted tips, advice, info

**NHS geography:** A common complaint/issue raised when doing a demo of the dashboard is that CCGs don’t exist any more and were replaced by ICBs. This is true, but the autism waiting times metric is still reported at a CCG level (technically now known as “Sub-ICB locations). That metric in future is likely to switch to ICB geographies, but for now we are using the most appropriate geographical boundaries. Ideally the NHS provider graphic would do something similar, but be aware that NHS providers can bid for and supply services across England; for example, health visiting services in areas of Teesside are provided by an NHS Trust in Harrogate.

**General shiny editing and errors:**

* If you get a “can’t subset elements past the end” error, you need an ungroup() before the mutate() call.
* Nested brackets are the worst part about Shiny, so take care when restructuring the dashboard UI as it’s easy to lose a bracket and wreck the whole thing.
* I made very limited use of git’s branch facility to test specific changes, so don’t worry about the branches other than ‘main’.