World Data:
a Web-Based Interactive Data-Reporting Application

Homer White
March, 2017

Table of Contents

Project Title and Description

The title of the project is:

World Data: a Web-Based Interactive Data-Reporting Application

The aim is to design and implement a proof-of-concept interactive data-reporting application that interfaces with a remote MySQL database. The application will be written in the R programming language using the web-app framework known as Shiny[1], and will interact with a convenient remote installation of the well-known `world`[2] database. Users will be able to customize interactively a variety of numerical summaries and graphical visualizations pertaining to the languages and demographic/economic characteristics of countries around the world.

Overview and Functionalities

Overview

The user-interface of the application shall be in the style of a dashboard with:

- Title Header
- Sidebar Menu
- For tabs linked to from the Sidebar Menu: a Main Content area consisting of:
    - a column of UI-widgets for interactive data analysis;
    - a column for display of output.

When the app is activated it sets up a "pool" of connections to the remote database. Initially there is only one connection, but as more users connect to the app, or as a single user makes many requests together, additional connections are opened, with the aim of ensuring that, when a user requests an action requiring a query to the database, an existing connection is used if at all possible. This reduces response-time. On the other hand a connection that has been idle for a specified period of time is closed. This minimizes use of the server's computational resources.

Once the pool is established the server makes two initial queries to the database, resulting in two tables (CountryLanguage and ScatterTable) that will be further

[1] https://shiny.rstudio.com/
[2] https://dev.mysql.com/doc/world-setup/en/

described below. These tables are available (without renewed queries) for the first user to connect and for all subsequent users, until all users disconnect and the app goes to sleep. (While it would be possible to construct specific SQL queries for *every* user action, this would result in excessively complex code as well as significantly degraded performance.)

Functionalities

We now turn to the major functionalities of the application.

Functionality 1: *Allow the user to rank the world's languages by number of speakers and to restrict the ranking to various regions.*

This functionality is provided by a "Language Popularity" tab that is shown when the user connects to the application.

- In the UI-widget area, the user selects a region from a dropdown menu.
- Application makes a SQL query to the database.
- The resulting table (languages in the selected region, ordered by total number of speakers within the countries of that region) is displayed. Each row corresponds to a language. Columns are: Rank of language, name of language, total number of speakers of the language.
- Data-table widgets appear above and below the table. These allow the user to:
    - Select the number of rows that are displayed on a single page of output
    - Navigate among pages (if there are more rows than will fit on a single page)
    - Re-order the rows according the values of each column.

Notes:

1. The data-table widgets will appear also in the displayed tables of Functionalities 2 and 3 below. For the sake of space we will not describe them again.
2. The data-table widgets work entirely within the browser from a table sent by the Shiny server. They work entirely with JavaScript in the browser and therefore do not involve requests to the server or queries to a database.

The following use-case diagram illustrates the primary part of the Functionality 1 (it does not include the data-table widgets).

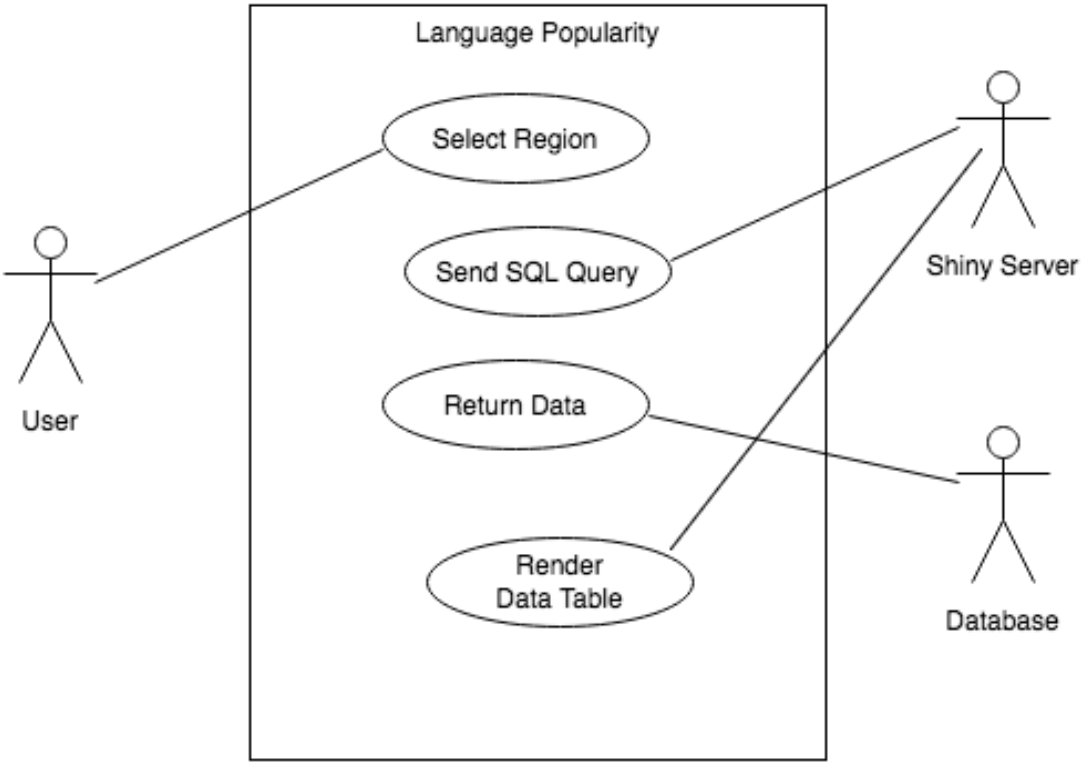

Figure 1: Use Case Diagram for Functionality 1.

It will be helpful to examine in greater detail how the Shiny server handles the user request.

A Shiny app has access to three objects: **input**, **output**, and **session**. The **session** object is outside the scope of the current discussion, but the other two objects deserve mention. The **input** object contains a number of properties whose values are determined by user actions in UI-widgets. The **output** object contains properties whose values correspond to items (tables, graphs and other information) displayed in the browser. The **output** objects are created and/or updated in response to updates of **input** objects. This is known as reactivity.

Figure 2 below indicates how reactivity works in Functionality 1.

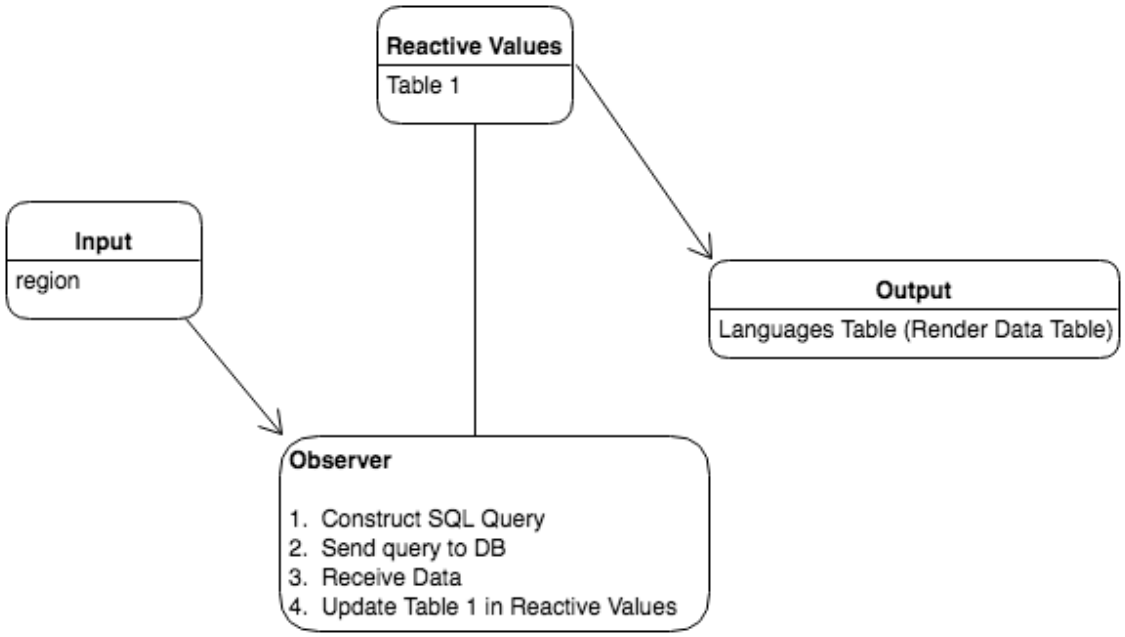

Figure 2: Shiny Server Reactivity (Functionality 1)

The server code includes an *Observer* (pictured in Figure 2 above) that is *reactively dependent* upon the input property "region." This reactive dependence, indicated by the arrow in the diagram, is established by a reference to the region-property in the Observer's code. The Observer is in the form of a function, but it does not return a value. It only has side-effects (i.e., it performs various actions). When the user selects a new region of the world, the region property is updated. This *invalidates* the Observer, and so the server (which checks every few microseconds for invalidated objects) runs the Observer code, resulting in the sequence of actions described in the figure, the last of which is to update the Table 1 property of a "Reactive Values" object. A reactive value is similar to an **input** property in that an update to it invalidates any object that has a reactive dependence upon it. The **output** object Languages Table takes a reference to Table 1, and so is invalidated as soon as Table 1 is updated. Within a few microseconds the server detects this and re-runs the code for Languages Table, thus rendering an updated data table for the browser to display.

Remarks:

1. The reader may wonder why an arrow of reactive dependency points toward the *dependent* object. This is a convention in Shiny diagrams (see, e.g., the RStudio article "Reactivity: an Overview") and is adopted due to the illusion, produced by the Shiny server's constant monitoring for invalidated objects, that a change in one objects "forces" a change in the dependent object.
2. There is a line but not an arrow between the Observer and the reactive value Table 1. This is because Table 1 is not reactively dependent upon the

observer; rather, the Observer updates the value of Table 1 whenever the Observer's code is run.

3. The reader might wonder why the Observer and the reactive value Table 1 are present at all. Why not simply have the Languages Table take a reactive dependency directly upon the "region" property and have its code deal with the SQL query, etc., and then render the data table to the browser? The reason is that if at some time in the future the developer wants to add another output feature that uses Table 1, he/she can simply write a new **output** object that takes a reactive dependence upon Table 1, without having to duplicate the code used to construct Table 1. We will see an example of this in Functionality 4 below.

Functionality 2: *Allow the user to select a language and rank countries according to how widely-spoken that language is in the country.*

This functionality is provided by a "Language/Countries" tab.

- User clicks on the Language/Countries link in the Sidebar Menu.
- The "Language/Countries" tab is displayed.
- In the UI-widget area the user selects a language from a dropdown menu (or types the initial letters of the language's name and uses auto-completion).
- The server returns a subset of the CountryLanguages table. This table is displayed in the output area. Each row of corresponds to a country in the world where the selected language is spoken. Columns are: Rank of country in terms of the percentage of its inhabitants who speak the language, name of the country, percentage of inhabitants who speak the language.
- Table widgets as described in Functionality 1.

Functionality 3: *Allow the user to select a country and rank its languages according to how widely spoken they are in that country.*

This functionality is provided by a "Country/Languages" tab.

- User clicks on the Country/Languages link in the Sidebar Menu.
- The "Country/Languages" tab is displayed.
- In the UI-widget area the user selects a country from a dropdown menu (or types the initial letters of the country's name and uses auto-completion).
- A table is displayed. Each row corresponds to a language spoken in the selected country. Columns are: Rank of language in terms of the percentage of its inhabitants who speak the language in that country, name of the language, and percentage of inhabitants in the country who speak the language.
- Table widgets as described in Functionality 1.

Functionality 4: *Allow the user to investigate associations (if any) between language-diversity within a country and its prosperity, modifying the definition of language-diversity and choosing from various measures of prosperity.*

This functionality is provided by a "Language/Prosperity" tab.

- User clicks on the Language/Prosperity link in the Sidebar Menu.
- The "Language/Prosperity" tab is displayed.
- In the UI-widgets area, the user:
    - selects as **response**-variable a measure of prosperity for the country (either the per-capita GNP of the country, or the mean life-expectancy in the country);
    - selects a **region** of the world;
    - enters a *commonality-criterion:* a value that sets a lower-bound on the **percentage** of inhabitants who speak a language in order for that language to be considered "commonly-spoken" in a country.
- The server constructs and sends a SQL query to the database, receiving determining a table consisting of countries from the selected region. The table has a column indicating the number of common languages in the country. The server returns a violin-plot grouping the countries according to the number of "common languages" they have. The vertical axis shows gives the selected prosperity measure. The horizontal axis gives the number of common languages in the country.
- The server also returns a table of summary data: for each groups of countries it shows: the minimum, first quartile, median, mean, third quartile and maximum of the selected prosperity measure for that group.
- The violin-plot is supplemented by jittered points that correspond to individual countries. Help-text invites the user to click near any individual point.
- On a click sufficiently near a point, the server uses the CountryLanguages table to return a one-row table identifying the country by name and giving its region, continent and year of independence. Another returned table shows that the languages that are common in that country, along with the percentages of the inhabitants who speak them.

It is worth looking at a Shiny server diagram for this case.

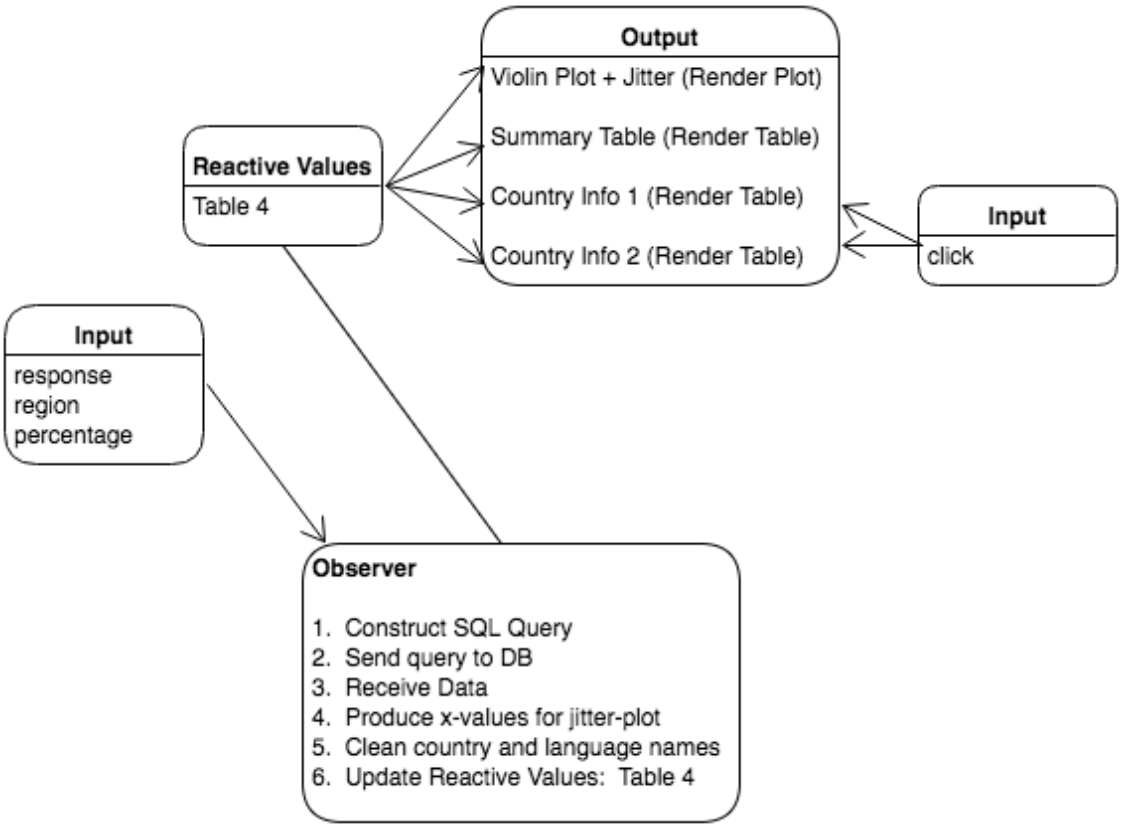

Figure 3: Reactivity Diagram for Functionality 4

In the diagram above we see that the Observer is reactively dependent upon three **input** properties: a change in any one of them will result in the code for the Observer being re-run within a few microseconds. Note that there are 4 **output** objects, each of which take a reactive dependency upon Table 4:

- the violin plot (supplemented with the jitter-plot);
- the summary table;
- the table that identifies the country whose point is clicked;
- the table that gives the common languages of the country whose point is clicked,

Of course, the latter two objects also depend upon the "click" property associated with the users action. This is required in order to identify the specific country corresponding to the clicked point.

Note: Almost all user input for this application involves selecting from a menu. However, in the **percentage** input field the user may type anything he or she wishes. Since this input helps to form a SQL query, we have to guard against the possibility of SQL-injection. Protection is rendered easy by the function **sqlInterpolate** provided by the R-package DBI that the application uses for to interface with the **shinydemo** database.

Functionality 5: *Allow the user to investigate associations (if any) between pairs of numerical variables recorded for individual countries.*

This functionality is provided by a "Scatterplots" tab.

- User clicks on the Scatterplots link in the Sidebar Menu.
- The "Scatterplots" tab is displayed.
- In the UI-widgets area, the user may:
    - select two variables from a list of four (population density, per-capita GNP, mean life-expectancy, percentage residing in the capital city);
    - use a range-slider to impose limit the values of the first-selected variable above;
    - use another range-slider to impose limit the values of the second-selected variable above;
    - use a checkbox to indicate whether or not a smoothing function will be applied.
- The server consults the ScatterTable and selects only those countries whose values fall within the ranges specified by the user. From this table it returns a scatterplot in which the x-axis represents the first-selected variable and the y-axis represents the second-selected variable. If the user checked the box for a smoother, then it also adds a loess-curve to the plot, along with an approximate 95%-confidence band around the curve.
- Help-text invites the user to click near any individual point.
- On a click sufficiently near a point, the server returns a table identifying the country by name and giving its region, continent and year of independence.

Functionality 5: *Allow the user to view documentation of the application.*

This functionality is provided by a "Documentation" tab.

- User clicks on the Documentation link in the Sidebar Menu.
- The browser opens a new tab showing a PDF of the documentation.

Since the project proposal, we have decided to add a functionality:

Functionality 6: *Allow the user to view the source code of the application.*

This functionality is provided by a "Source Code" tab.

- User clicks on the Source Code link in the Sidebar Menu.
- The browser opens a new tab showing the Git Hub repository for the source code of the application.

This concludes our specification of the specific requirements associated with each functionality of the application under design.

Database Design

The database used in the project was set up by RStudio as a sample online tool to assist in writing tutorials on database access in R. It's a MariaDB database—MariaDB is a fork of the MySQL project, so it is a traditional relational database in most respects. The name of the database is **shinydemo** and it may be found at the following URL:

```
shiny-demo.csa7qlmguqrf.us-east-1.rds.amazonaws.com
```

The structure of **shinydemo** is shown in the following Entity-Relationship Diagram:

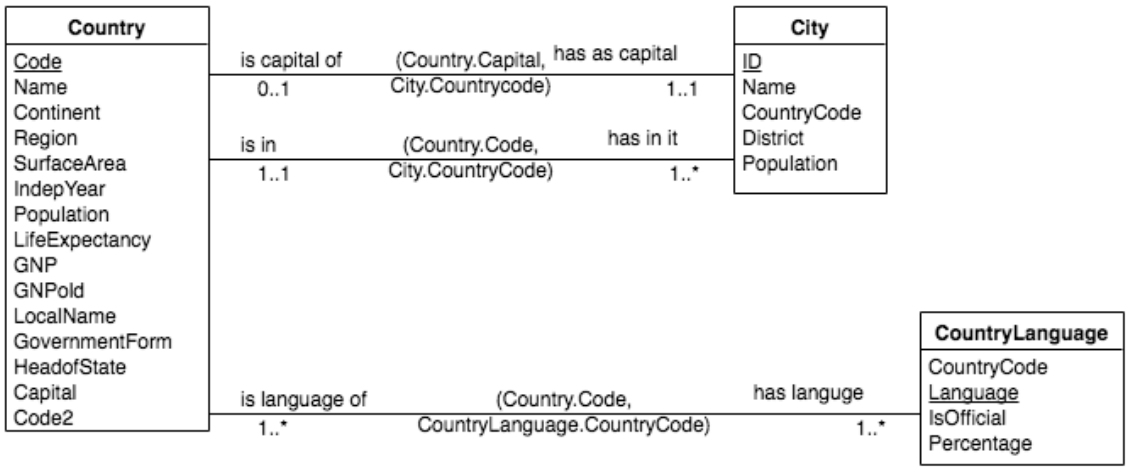

Figure 4: ERD for the shinydemo "World" database

User-Interface Mockups

The code for a Shiny app includes the definition of two major functions: one that controls the user-interface and another that controls the server. In small applications a single person is liable to write both the ui and server functions, but in larger applications the work can be divided: a person with front-end skills can write the ui function, along with a "skeleton" server function. The resulting code may then be deployed as an app that provides a mock-up of the user interface.

I have adopted the above approach in order to demonstrate mock-ups for the application under design. The mock-up app may be accessed at the following URL:

https://homer.shinyapps.io/mockups/

Implementation

The application was successfully implemented, and it may be viewed at the following URL:

https://homer.shinyapps.io/countries/

The reader will note that the app itself contains a link to the GitHub repository containing its source code, but the link is also provided here, for convenience:

https://github.com/homerhanumat/countries

Source code for the mock-ups application resides on the **mock** branch of the repository.

The application should support multiple simultaneous users, but since the **shinydemo** database permits no more than 16 open connections from a single client, the number of possible simultaneous users of the app is far less than what the Shinyapps platform would generally allow.