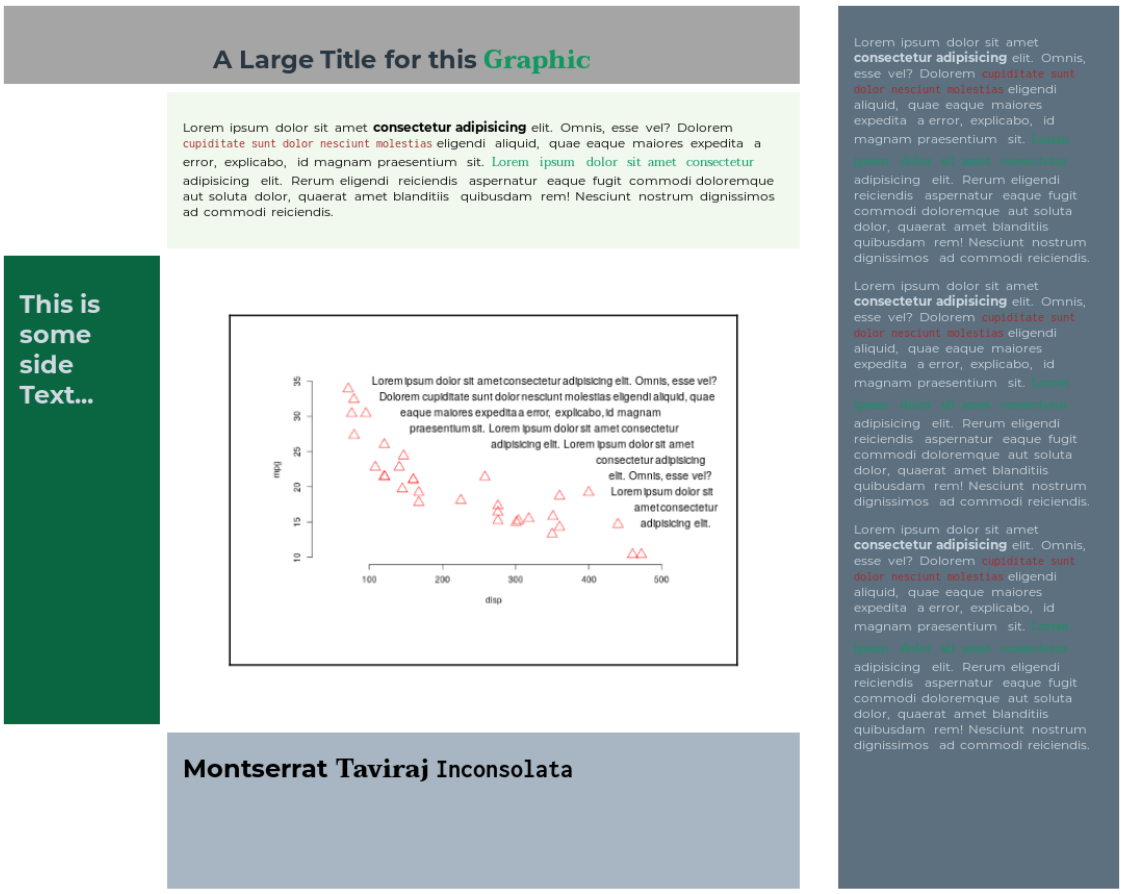
Can Web Technologies Help R Generate Print Quality Graphics?


A Large Title for this Graphic
This is some side Text...
Montserrat Taviraj Inconsolata
mpg
disp

Lorem ipsum dolor sit amet **consectetur adipisicing** elit. Omnis, esse vel? Dolorem `cupiditate sunt dolor nesciunt molestias` eligendi aliquid, quae eaque maiores expedita a error, explicabo, id magnam praesentium sit. Lorem ipsum dolor sit amet consectetur adipisicing elit. Rerum eligendi reiciendis aspernatur eaque fugit commodi doloremque aut soluta dolor, quaerat amet blanditiis quibusdam rem! Nesciunt nostrum dignissimos ad commodi reiciendis.

CompSci 791: Research Paper
Faculty Advisor: Dr. Paul Murrell

Author: Kane Cullimore

Submitted on
November 23rd, 2020

Abstract

The theme of this research paper is captured in the titled question ***"Can Web Technologies Help R Generate Print Quality Graphics?"*** This question originated from Dr. Paul Murrell's [1] ongoing efforts to extend R's graphic system. This research is focused on extending an experimental R package, called the layoutEngine [2] [3], in exploration of this underlying question.

The R programming language [4] has a powerful graphics system [5] but it has several limitations within the scope of Print Quality Graphics. The layoutEngine attempts to provide this functionality with a non-standard approach. It is designed to link R's graphic system to a Web Browser and thereby tap into the rich and well established technologies of HTML, CSS and JavaScript.

This research includes the development of a major component of the layoutEngine in order to overcome several existing limitations of the existing implementations. Two versions of the same component were performed to explore different implementations of requirements. These versions are called (i) the **RSelenium Backend** and (ii) the **NodeJS Backend**. The development efforts show promise in both directions however the NodeJS version appears to offer the most promise for future work.

The research paper will explain the *layoutEngine approach* to extending R's graphic system. It will then review the existing solution design and present several key limitations. The paper then steps through the solution design of the RSelenium and NodeJS Backends and present how these implementations improve upon the existing designs. It should be noted the comparisons made between each implementation are qualitative in nature as the primary objective was to explore alternative solutions as opposed to performance enhancements which are easily measured. A review of both newly developed Backends compared to the existing implementations is provided to conclude.

Research Development Content

The following GitHub repositories and Docker images were developed as part of this research and are submitted as a practical accompanyment to this report.

Research Files:

- GitHub: Report and Supporting Content [6]
- Docker Image: Reproducible Report [7]

RSelenium Backend:

- GitHub: layoutEngineRSelenium R Package [8]

NodeJS Backend:

- GitHub: layoutEngineExpress R Package [9]
- GitHub: NodeJS Backend Server [10]
- Docker Image: NodeJS Backend Server [11]

Contents

1 Introduction

The **R programming language** [4] is a popular open-source statistical analysis tool. The language has many packages that support sophisticated statistical techniques. Many of these rely on graphical output to communicate results. A strong appeal of this programming language is the ease at which its core **graphics system** [5] generates graphical output that is both accurate and effective at communicating this type of information.

Few open-source alternatives offer an equivalent set of sophisticated statistical methods married with a flexible and powerful graphics system. The Python programming language is a strong competitor. However, its focus is more as general purpose programming language with fewer specialized statistics packages that generate these types of graphic types with equivalent rigor.

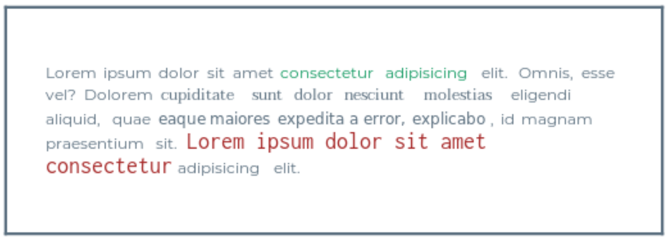
The number of specialized applications R is used within has grown with its popularity. One such use-case is the incorporation of the statistical graphic within published articles. While the raw dots and dashes used to generate these graphics is of sufficient digital quality there are several fundamental publishing requirements which are not supported.

2 Problem Description

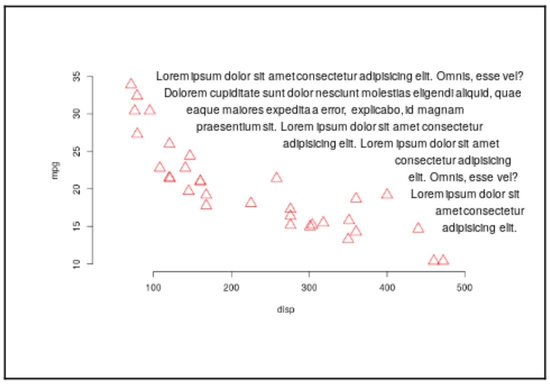
The publishing industry has a long history dating back to the 15th century when movable type printing was first invented. As the industry integrated within digital media it has brought with it a system of long-standing expectations for content. As a result, digital publications often have a myriad of requirements for graphics which are referred to in this report as **print quality graphics**.

Some of the more important requirements include **(1)** writing system and font specifications, **(2)** document layout and typesetting, **(3)** sophisticated content rendering, and **(4)** control over output resolution and file format.

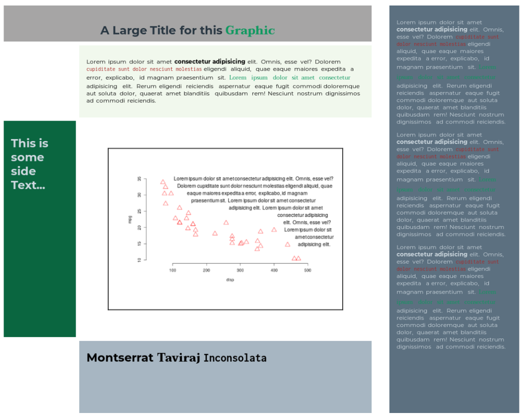
R users have a powerful tool for generating statistics based graphics but they will struggle to support many of these requirements. Several examples are presented below which show features which are not currently possible with the core graphics system.


Lorem ipsum dolor sit amet consectetur adipisicing elit. Omnis, esse vel? Dolorem cupiditate sunt dolor nesciunt molestias eligendi aliquid, quae eaque maiores expedita a error, explicabo , id magnam praesentium sit. Lorem ipsum dolor sit amet consectetur adipisicing elit.

Example 1: Using Multiple Font Types in the Same Graphical Component

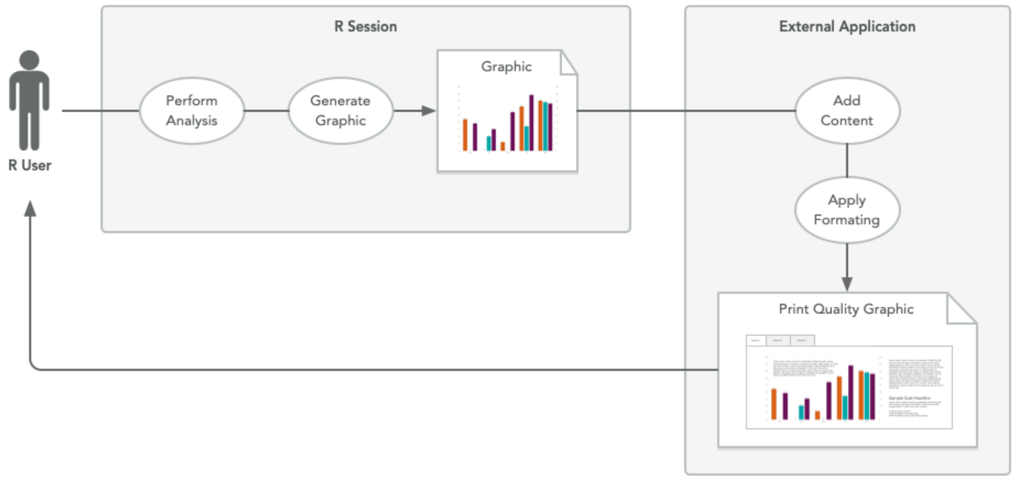


Example 2: Embedded Graphics With Automatically Wrapped Text


A Large Title for this Graphic
This is some side Text...
Montserrat Taviraj Inconsolata

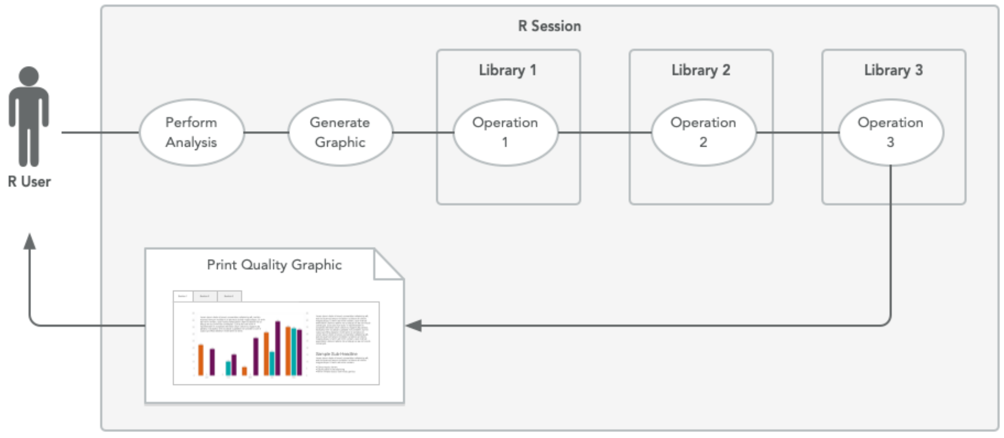
Example 3: Complex Layouts Of Text And Graph Components

An existing solution to produce print quality graphics is to modify the R graphical output with external tools such as LaTeX or *Adobe Illustrator*. The user must either be proficient in both environments or have a specialist available to help.



External Applications Used to Produce Print Quality Graphics

Another solution would rely on several existing R packages. Such packages offer bits of this functionality ad hoc. The user would remain within the R ecosystem but might need several packages depending on the publishing requirements. This modular feature-set composition is the *standard R approach* to extend its functionality.



Standard R Approach to Produce Print Quality Graphics

This research explores an alternative approach, referred to as the *layoutEngine approach*, which offers a general purpose solution to generate print quality graphics from within the R ecosystem. This approach is encapsulated by the layoutEngine R package which incorporates web technologies to extend the functionality of the R graphics system.

The motivation of this approach is based on the understanding that web technologies and modern browsers have long supported the special needs of the publishing industry. As a result, there is much functionality to be gained by leveraging this technology. The layoutEngine intends to create access to a full-suite of industry leading functionality. This hinges on whether it can successfully utilize the browsers graphics system in an easy to use R package with accessible dependencies.

Please note that this research does not address the relative performance and functionality difference between the *layoutEngine approach* and the *standard R approach*. Rather, it *first* explores the efficacy of this *layoutEngine approach*, and *second*, attempts to improve upon the implementation of the existing layoutEngine package to address several existing limitations.

3 The layoutEngine R Package

The intent of the **layoutEngine** [2] R package is to extend R's graphics system by adopting functionality available in web technologies. Its core functionality is to act as an interface between R and a web browser to provide access to the rich feature set. The package is available to review via the layoutEngine GitHub repository [3] however it is still in development and not yet available in **CRAN** [12].

3.1 The Standard R Approach

The *standard R approach* to extend its functionality is paramount to the success of open-source programming languages. Available packages are freely shared within the community which can be loaded into the scope of an environment to gain specific functionality. If a broader feature set is needed then several packages are loaded which acts as a modular system. There are many advantages to this approach and it is a key reason why these languages and user communities have thrived.

Several **available CRAN packages** [13] offer functionality which meet specific publishing requirements. Many of these packages are well executed and perform admirably. The *ggtext* package [14] enables multiple font types to be specified in the same graphic. The *patchwork* package [15] offers a similar functionality specifically for arranging several ggplots with claims of increased simplicity and flexibility. In addition, the base package *grid* [16] has a `layout` function which creates a `Grid` layout object [17] that enables plots from different systems to be arranged together. Many other packages extend R towards the realm of print quality graphics but no general purpose solution exists at this point in time.

The standard R approach might eventually succeed in offering a general purpose solution to generate print quality graphics. However, several difficulties exist would first need to be overcome. First, the publishing requirements represent an extensive set of functionality. In addition, the variety of graphical output this must operate with is also large. As a result, the task of coordinating a suite of purpose-built packages that is flexible enough to cover all scenarios would be significant. This is both difficult from both a developer and user perspective due to the large number of package specific syntax to navigate.

3.2 The layoutEngine Approach

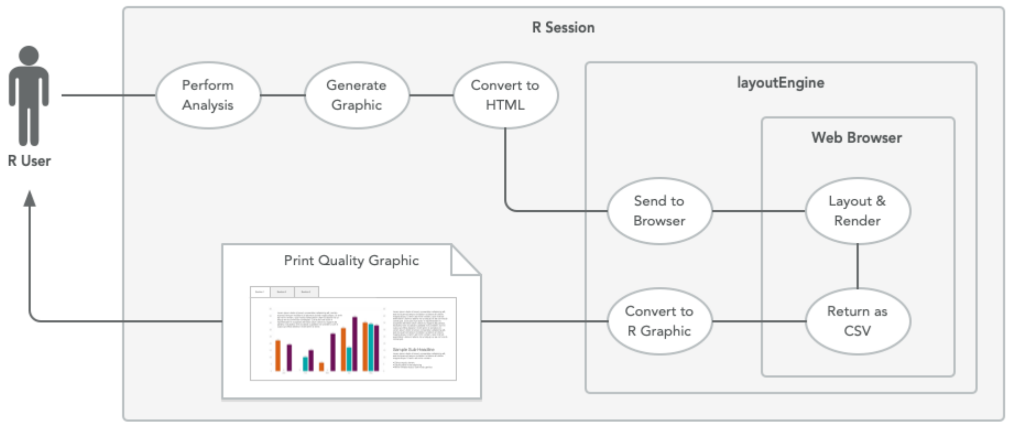
The *layoutEngine approach* differs with the way it extends the functionality of R. It acts as an interface between the graphics system in R and another technology ecosystem: the web browser. This general purpose solution attempts to adopt from an industry that has a long history supporting publishing requirements so as to avoid reinventing the wheel.

This approach bypasses the need to build a complex general purpose solution from the ground up and instead employs existing tools. These tools are part of the web-based languages (HTML, CSS and JavaScript) and R packages which can be used generate HTML from R objects. The table below identifies some of these technologies which can be used to generate a print quality graphic.

Functionality	**Technology**
Layout and Typesetting	HTML elements styled with CSS Grid or Flexbox
Font Type and Styling	CSS @font-face with traditional CSS styles
Text Wrapping	Standard HTM, CSS shape-outside
R to HTML Conversion	xtable, formattable, htmltools, gridSVG, rmarkdown, etc...

The simplified diagram below demonstrates how the layoutEngine is used for a general use-case. The process would include the following:

1. The graphic is first generated as an R object
2. This is converted to HTML where some additional definition could be added
3. The HTML based data is transferred and loaded into a web browser
4. The browser's layout and rendering engine generates the desired graphic in the browser window
5. A JavaScript function is then executed to separate and calculate the position of each component on the page
6. This data is then sent back to R in CSV format where the layoutEngine will convert it to a R readable graphic object
7. This can then be displayed in the R graphics window or sent directly to an image file



Produce Print Quality Graphics Using the layoutEngine

3.2.1 Benefits

There are several key benefits to the *layoutEngine approach* that are the motivating drivers of this exploration.

The most important of these is that the HTML and CSS languages are well suited to managing attributes of elements **within** the graphic. The intention is to not just drop an R graphic within the middle of some HTML as a "dumb blob" but rather to further enhance its styling and content. It is the mechanism for R users to *programmatically* control aspects of the graphic in a precise and repeatable manner that is of most interest.

A second key advantage is that web technologies serve a large economy that has demanded a robust feature set and accessible programming paradigms for several decades. Part of this economy, one of which is the publishing industry, cares greatly about the aforementioned features that fall within Print Quality Graphic. If the approach is successful we can hope to gain the following advantages:

- Large community of developers that are well versed in the web languages (HTML, CSS and JavaScript)
- Active support and advancement of standards, new features and extension libraries
- Large amount of references and resources for new developers to become proficient quickly
- Sophisticated layout and rendering engines within modern browsers are continually being updated to support the latest web standards

Lastly, there are already several R packages that generate and manipulate HTML and SVG content. The hope is that an R User can take advantage of these packages to help improve the efficiency the can generate the needed HTML to work with the layoutEngine. Some of these packages are Knitr, xtable and htmltools.

3.2.2 Limitations

The *layoutEngine approach* offers some compelling advances but there are several limitations that should be recognized.

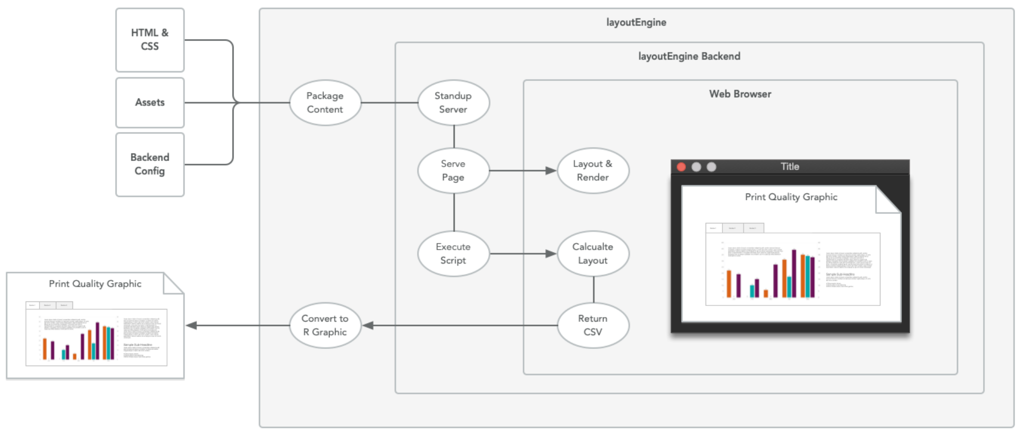
The first and perhaps the most critical is that R users need to enter HTML based code in the layoutEngine functions. This requires the use of an available R packages to create HTML from their R code. If the want to use anything semi-customized or unique they would be required to generate that directly within HTML syntax. While these web languages are common in the industry it does require the user to either already be proficient in these or willing to learn.

The complexity of the interface between the R Session and the web browser is substantial. Since the browser has been built for use with the public internet there are several security features to get around. The R session cannot just load a local file into the browser and start using its layout and rendering engines. Instead, a web server is needed to host a web page on the local machine's network thereby serving the page in the manner expected by the browser. This requires several additional technical requirements that add to the complexity of the basic concept.

Another important limitation is based around the state of the implementation of this layoutEngine R package. Since it is very much in a prototype phase there are several missing features and incompatibilities between the web languages and what is able to be converted back to an R graphic object. As such, there might be some frustration or lack of completion due to these gaps.

3.3 Solution Design

The layoutEngine solution design is comprised of two components as shown in the figure below. The layoutEngine is configured to interface with one of several available **layoutEngine Backends** [2]. The layoutEngine acts as the interface for the R user while the layoutEngine Backend is the interface between R and the web browser. This solution was chosen as much of the complexity exists in the Backend. This abstracts much of the complexity away from the user and allows various Backend designs to be swapped out with little impact to the main layoutEngine interface.



layoutEngine Solution Design

3.3.1 layoutEngine R Package

The layoutEngine is responsible for handling all the web-based data to send to the layoutEngine Backend. It must also take the returned data from the Backend and render this within the R graphics display. A major piece of this involves locating the correct fonts within the host machine to ensure the browser calculations are made using the exact font specifications.

Functionality:

- Interface for R users to execute commands and configure the Backend
- Handling of web page content (HTML, CSS, Fonts, Assets)

- Call of Backend primary interface
- Handle returned layout calculation
- Rendering of new content within R Graphics Display

Challenges:

- Ensuring ease-of-use for R users by tolerating various web-based content types and formats
- Cross-platform access to system fonts
- Support conversion of web-based graphic data to the R graphics display

3.3.2 layoutEngine Backend R Package

The layoutEngine Backend is where much of complexity exists required to interface with a web browser. Since there are many possible ways to implement this functionality it is contained within a separate R package. The main purpose of the Backend is to serve the R graphic to a browser, execute the layout calculation script and return data to the layoutEngine. The Backend contains the primary challenge of the *layoutEngine approach*.

Functionality:

- Locate and manage a modern web browser session
- Send and receive data between a R session and web browser
- Query and modify the web-page DOM

Challenges:

- Variability in Host Machine
  - Cross-platform system calls for macOS, Windows and Linux
  - System dependencies and installation requirements
- Web Browser Interface
  - Server implementation
  - Bidirectional communication between R and the browser

4 layoutEngine Development

The viability of the *layoutEngine approach* is still being explored and it is the layoutEngine backend where the majority of the limitations reside. There are several existing layoutEngine Backends which are presented in this section. Each has certain benefits and limitations. At this point in the exploration of the *layoutEngine approach* none are good candidates for a *production* variant.

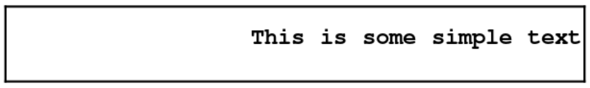
The focus of the subject research focuses primarily on developing some alternatives to these existing implementations of layoutEngine Backends. This includes the development of two new layoutEngine Backends which aim to incorporate a **Target Feature Set**. This set was identified as being paramount to the needs of a production ready solution. One of these Backends relies on a **Selenium** [25] server hosted within a Docker container [26]. The second Backend is a custom **NodeJS** [27] server also hosted within a Docker container.

This section outlines the primary requirements for these development efforts including: **(1)** Component Interfaces, **(2)** Minimum Technical Requirements and **(3)** the Target Feature Set. The existing Backends are presented and qualitatively scored against these requirements as a means of comparison of the new development effort.

4.1 Component Interfaces

Development for a layoutEngine Backend requires meeting the interface requirements for the layoutEngine package itself. In the example below, the Backend is specified and then the layoutEngine **`grid.html()`** function is called with the specified arguments. Since the Backend is isolated, this *R user interface* remains the same for all Backends.

```
## Load packages for layoutEngine and Backend and set the Backend
package(layoutEngine)
package(layoutEngineRSelenium)
options(layoutEngine.backend=RSeleniumEngine)
## Basic example supplying HTML, FONTS, CSS, ASSETS
HTML <- '<div class="content"><h1>This is some simple text</h1></div>'
FONTS <- cssFontFamily("FreeMono")
CSS <- '.content {font-family: FreeMono; border: solid; width: 800px; height: 100px; display: flex;
 ↪  justify-content: right;}'
## Call layoutEngine
png("essential-example.png", width=800, height=100)
grid.html(html=HTML, fonts=FONTS, css=CSS)
dev.off()
```



layoutEngine Output: "essential-example.png"

The *Backend interface* is contained within the **`engine.R`** component of all the Backends. This file specifies a **Layout** and **fontFile** function that is used to instantiate a layoutEngine Backend Engine object using the **`layoutEngine::makeEngine()`** function. The requirements and flexibility of this object is documented within the **`makeEngine()`** help page [28].

The **`layout`** argument is the minimum requirement needed to instantiate Backend object. This function should perform take the supplied web-page data and then layout the page with a layout engine. This engine is expected to be provided within a typical browser but as seen with the PhantomJS and CSS-Box Backend solutions can also be a atypical implementation of a layout engine.

Once the web-page layout has been performed within the browser (or equivalent), the next requirement is the layout is captured within a R readable form. This is a critical piece of the layoutEngine functionality. For this research, the script that executes this step was taken from the DOM Backend. In general, this JavaScript script updates the web-page DOM to isolate all text characters within a single bounding box. Once each element has its bounding box, the locations and content for each box is exported from the DOM and transformed into CSV format.

The layout CSV is then transferred back to the layoutEngine primary package where it redraws the graphic within the R graphic display. For all of this to work well, each component must be defined and tracked through this entire process. Of special concern is that of the specified fonts. Since all fonts vary dramatically in their sizes, the actual font files must be identified on the host machine and then transferred to the browser where they are incorporated into the web-page using the `@font-face` CSS feature.

The layoutEngine Backend has much flexibility in the way it can be implemented provided this core functionality is provided. In general, these minimum requirements of the `layout` function include:

1. Start up a server used to host the web-page
2. Find the specified font files within the host system
3. Copy asset files to the server (fonts, images, embedded HTML, etc)
4. Open a web browser
5. Serve the web-page
6. Execute the layout calculation script within the browser
7. Send the returned CSV back to the layoutEngine

4.2 Minimum Technical Requirements

The layoutEngine Backend must meet several key technical requirements to adequately perform the necessary functions as outline in the previous section. These requirements can be addressed in a variety of ways as is evident in comparing the existing Backends later in this section. These Minimum Technical Requirements have been categorized as-follows to improve the ease of comparison.

Locally Hosted Server:

- ▶ Provides a mechanism to communicate with a web browser and access the full capability of the layout and rendering engines
- ▶ Supports live refresh of the web page upon when changes are made is necessary to give the R user a similar experience they're used to with the R graphics display. This prevents the user from having to push a 'refresh' command to see the changes.

Browser Integration:

- ▶ Use of a modern web browser to ensure ability to utilize the best features of HTML, CSS and JavaScript as they become available
- ▶ Interactions with browser can be implemented in a traditional protocol such as HTPP or Web Sockets. It can also be implemented with a purpose built API such as Selenium or Google's Puppeteer [29].
- ▶ Maintain a single web browser instance for each layoutEngine call to prevent multiple browsers from sending layout calculations back to R

Bidirectional Communication:

- ▶ The backend must support a browser compatible communication protocol
- ▶ Data must be transferred between R and the browser in both directions. The backend must first send the web-based data to the browser and then on the completion of a layout calculation, send a text-based message back to R.
- ▶ Data can be of the following types
  - ▷ To the browser
    - ⋆ Asset files such as images and font files to the hosted webpage from the host machine
    - ⋆ DOM components from the R layoutEngine call
    - ⋆ Call to execute the layout calculation script
  - ▷ From the browser

    ⋆ Either request or accept the layout calculation CSV as a file or in memory data

System Interface:

- A key piece requirement of the layoutEngine is that it can access the host machine's system fonts. This requires the layoutEngine Backend to execute system calls to locate and retrieve such font files.
- Any virtual machine based solution must be able to access the graphics system of the host machine to run a browser in a traditional manner (i.e. not in headless mode)

4.3 Target Feature Set

A Target Feature Set is defined below to include features that are identified as being paramount to the needs of a production ready solution. These features limit the barrier to successfully utilizing the layoutEngine approach for a general R user. If these can't be met, the approach would be too cumbersome for most users to adopt.

Cross Platform:

An emphasis is placed on new layoutEngine Backends to support all three major platforms (Linux, MacOS and Windows). While the existing Backends prove the viability of the layoutEngine functionality it is deemed absolutely necessary for the package to be easily used on all platforms. There is little chance the package would be found useful across the industry if it were only available on Linux platforms. A primary reason for this being that a majority of users are on either Windows or MacOS. The challenge is mainly based on the installation dependencies and host graphics device usage.

Simple Dependencies:

Secondary to cross platform support, the backend must also have relatively simple installation requirements for all platforms. The intention here is to improve the user experience by making the installation as easy as possible. In addition, with fewer requirements there is less opportunities for future incompatibilities to arise.

- Few dependencies on other development teams for critical components that might either create bugs or prevent the support of the latest web technologies
- As simple implementation as possible which:
  - Reduces the weight of the installed package
  - Enable easy future support or extensions by others
  - Provide as few opportunities for issues, bugs, etc and improve the maintainability and comprehension by other contributors to the project

Industry Support:

It is critical that any technologies that are incorporated into the layoutEngine backend have development support into the future. The more common and widely used such technologies are the less likely there will be technical issues as other parts of the ecosystem advance.

Modern Web Standards:

It is preferred the backend design is able to support modern web standards for *Web Design and Applications* as defined by W3C [24]. If the latest and greatest standards are not fully supported then an acceptable lag of 1 to 2 years from the most recent release. This feature should be considered as relatively important as many users will be turned off from too much lag between what is seen as industry standard versus cutting edge.

Visual Feedback:

It should be considered valuable to have access to a live browser for several reasons. Although headless browsers might be considered more easily implemented there is significant value in being able to see the graphical output within the browser. For example, the user can see quickly identify any discrepancies between in supported web technologies between the browser and R graphics display.

4.4 Existing Backend Designs

There are three layoutEngine Backends available for use with the layoutEngine at the time of this research. These Backends successfully demonstrate the viability of this approach.

4.4.1 DOM Backend

The DOM Backend [18] is fully contained within R and employs only R packages to gain the minimum technical requirements.

Benefits

- Built entirely within the R ecosystem
- Access to latest web browser and therefore latest HTML, CSS and JS specs

Limitations

- Default browser opens every call

Minimum Technical Requirements

- Locally Hosted Server:
  - The R httpuv package [19] is used to create a static web server for web page hosting.
- Browser Integration:
  - Once the server has been initiated and the layoutEngine grid.html() function has been called, the default browser on the host machine is automatically launched and navigated to the hosted page at `http://localhost:port-number`. The port number is randomly generated for each layoutEngine grid.html() function call to reduce the likelihood of conflicting port assignment.
- Bidirectional Communication:
  - The R httpuv package is also used to create a Web Socket server which creates a bidirectional communication protocol with the local default browser.

Ratings for Target Feature Set

- Cross Platform: **Medium**
  - The DOM Backend has only ever been used and tested on a Linux machine. It uses specific Linux system calls of which several would fail in other operating systems. The rating is set at Medium since much of this could be re-engineered to work on other platforms.
- Simple Dependencies: **Medium**
  - The primary dependencies for the DOM Backend consist of the Linux OS and several R packages. Outside of the OS requirement the dependencies are relatively simple for an R User.

- Industry Support: **Low**
  - The DOM Backend is custom built on very primitive server based methods provided by the httpuv R package. While the httpuv package is actively maintained by members of the RStudio team the DOM package itself contains much of the necessary functionality that would need to be extended and maintained if this Backend was used as a production ready design.
- Modern Web Standards: **Medium**
  - The DOM Backend relies on the users local browser which is the best case scenerio for ensuring the latest web technology standards are available for use. However, the implementation of the DOM Backend uses R wrappers around JavaScript DOM scripting methods. As such, these would have to be maintained and extended to accommodate any significant changes in web standards.
- Visual Feedback: **High**
  - The Backend uses the local browser with full visibility of the web page content.

4.4.2 PhantomJS & CSSBox Backends

The PhantomJS and CSSBox Backends have been combined here since they are very similar in their implementations as it pertains to the following requirements. Both utilize non-browser layout engines. The PhantomJS Backend [20] uses the PhantomJS [21] scriptable headless browser to interface directly with the web page DOM. The CSSBox Backend [22] uses the Java based CSSBox [23] headless browser to interface directly with the web page DOM.

Benefits

- Implementations have single dependencies on their respective HTML layout engine technology
- Headless browser API greatly simplifies the requirements to host a web page and communicate with a typical browser

Limitations

- No option for visual browser use
- Both HTML layout engine technologies suffer from development support issues
- Based on older layout engines so behind on latest web standards

Minimum Technical Requirements

- Locally Hosted Server:
  - Both Backends have standalone layout engines that don't necessitate the requirement to access the page from a traditional server. As such, neither require a server to be running which is a significant benefit to both of these designs.
- Browser Integration:
  - The "browser" in these cases refers only to the layout engine component of each of these headless browsers. Again, this approach simplifies the need to integrate with an open browser R would only need to access each of these systems that are running on the host machine. The requirement instead is that the R user would have to have either system installed and configured appropriately on their host machine.
- Bidirectional Communication:
  - Both have their own API that can be access programmatically from R. Each API implements the

DOM scripting methods that allows access to the DOM to query and edit.

Ratings for Target Feature Set

- Cross Platform: **Medium**
  - Similar to the DOM Backend, both of these Backends have only ever been used and tested on a Linux machine. Both use specific Linux system calls of which several would fail in other operating systems. Both the PhantomJS and CSSBox can be configured to run on all major operating systems so they should not pose any limitation to this target feature. The rating is set at Medium since much of this could be re-engineered to work on other platforms.
- Simple Dependencies: **Low**
  - Both Backends require the installation and configuration of these independently maintained tools. CSSBox is especially difficult as it relies on a compatible Java JDK to be available on the host machine. This dependency is quite limiting for a general purpose tool.
- Industry Support: **Low**
  - The PhantomJS developer team have already declared the tool is no longer supported. The CSSBox is still being maintained however the last version update occurred in November of 2019.
- Modern Web Standards: **Low**
  - Since both tools lack adequate support to keep their API's and layout engines compatible with the latest web standards.
- Visual Feedback: **Low**
  - Neither support visual feedback as they're both headless browsers and therefore can't provide a live view of the browser screen.

4.4.3 Qualitative Comparison

The following list outlines how each layoutEngine Backend was ranked against the target feature set:

- **Cross Platform:** Compatibility with all major operating systems (Linux, MacOS and Windows)
- **Simple Dependencies:** Simple installation with few dependencies that are consistent across platforms
- **Industry Support:** Robust industry support of any incorporated tools, technologies and standards
- **Modern Web Standards:** Support for modern web standards including HTML, CSS and JavaScript
- **Visual Feedback:** Ability to view graphics within live browser

Limitation	DOM	PhantomJS	CSSBox
Cross Platform	Med	Med	Med
Simple Dependencies	Med	Low	Low
Industry Support	Low	Low	Low
Modern Web Standards	Med	Low	Low
Visual Feedback	High	Low	Low

5 RSelenium Backend

The RSelenium Backend has been developed with the use of two preexisting technologies. The first is the **Selenium WebDriver** [25] which is a robust browser automation tool. The second is the **RSelenium R package** [30] which provides an interface to the Selenium WebDriver from within R. The Selenium WebDriver is used from supported Docker images which they host on their Selenium Docker Hub site [31].

This section first presents the solution design of the layoutEngine Backend using Selenium. This is then followed by an evaluation of the design based on the Minimum Technical Requirements and Target Feature Set defined earlier in the report.

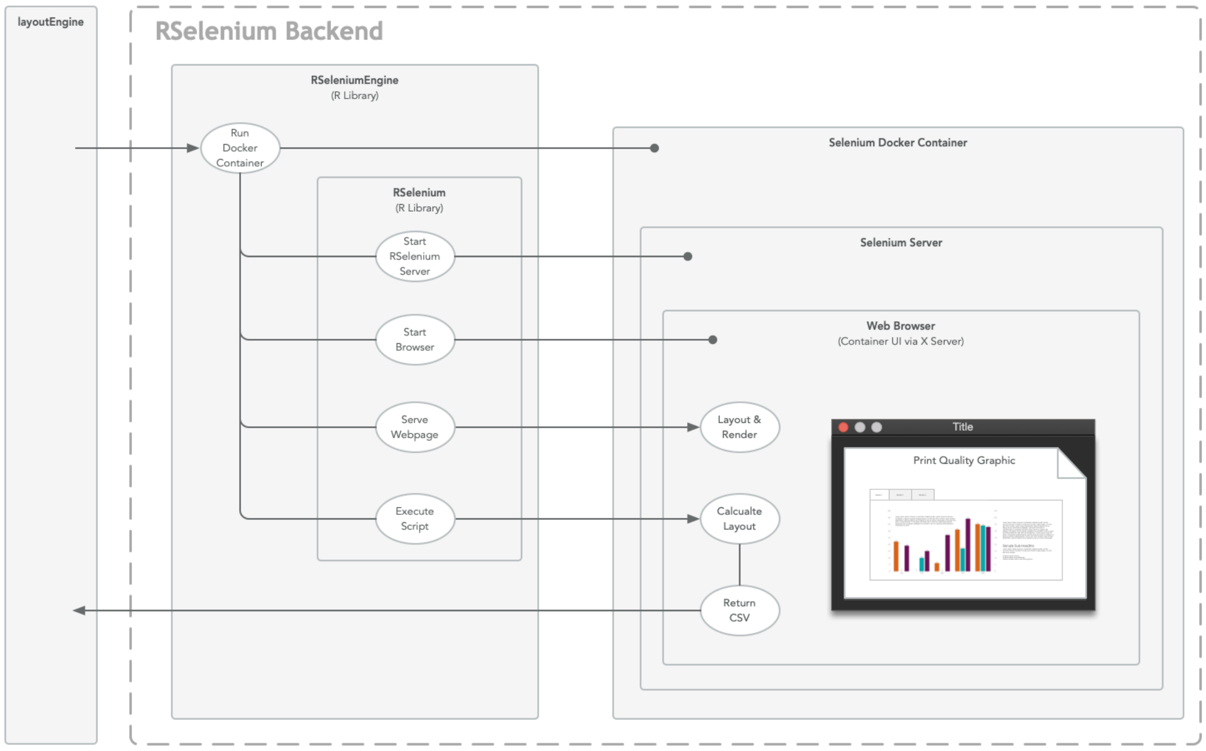
The primary interest in using Selenium is that is a powerful and popular tool that offers a more robust platform to control a web browser. The fact the RSelenium package allowed for easy use of this tool off-the-shelf further increased its appeal. While the benefits for this solution design are significant there are also several limitations (or downsides) to its use. The motivation behind the RSelenium Backend package includes the following:

- Selenium Server
    - Robust solution that offers a large set of ways to control a browser
    - Strong technical community with evidence of future support due to wide industry use
    - Support of docker containers
    - Docker container includes controlled web browser types which improves consistency
    - Direct access to the webpage DOM via the Selenium server commands
- RSelenium
    - R interface to Selenium Server already exists
    - Tests shows relatively simple controls

5.1 Solution Design

5.1.1 Primary Components

- RSelenium R Package
- Selenium Docker images
- Docker container
- Browser launches from docker container (good and bad)



RSelenium Backend Design

5.1.2 Function Call Relationships

1. RSelenium Backend Arguments

```
- url='127.0.0.1'
- portRS=4444
- portClient=4444
- network='bridge'
- shm_size='1g'
- browser_type='firefox'
- headless=FALSE
- image_request=NULL
- fresh_pull=FALSE
```

2. layoutEngineRSelenium Backend Setup

```r
package(layoutEngine)
package(layoutEngineRSelenium)
options(layoutEngine.backend=RSeleniumEngine)
## Firefox Selenium docker image
firefox_build <- "3.141.59-20200525"
firefox_image <- paste0("selenium/standalone-firefox-debug:", firefox_build)
## Chrome Selenium docker image
# chrome_build <- "3.141.59-20200525"
# chrome_image <- paste0("selenium/standalone-chrome-debug:", chrome_build)
## Select which image to test
test <- list(name="firefox", image=firefox_image)
# test <- list(name="chrome", image=chrome_image)
```

3. Docker Container Component
   The Selenium Docker container call **`dockerContainer()`** will start up the Selenium Docker container per the argument settings. It also has methods to get status and stop the container.

```
## User settings for component test
settings <- list(url="127.0.0.1", portRS=4444, portClient="4444", network="bridge", shm_size="1g",
                 browser_type=test$name, headless=FALSE, image_request=test$image, fresh_pull=FALSE)
## Setup RSelenium Backend Docker container based on user settings
container <- dockerContainer(settings)
## Run the RSelenium docker container
container$run()
## Get RSelenium docker container status
container$getInfo()
## Stop and delete the RSelenium Docker container
container$close()
```

4. Selenium Server Component
   The Selenium server call **`rSServer()`** will start up the Selenium Docker container with the **`dockerContainer()`** call shown above. It will also setup the RSelenium server hosted within the Docker container with the **`open()`** method. It also has methods to get status and stop the container.

```
## Setup RSelenium Backend Selenium server hosted on the docker container
RSServer <- rSServer(settings)
## Get Selenium server status
RSServer$getStatus()
## Open a browser instance hosted within the Docker container
RSServer$open()
## Close RSelenium Backend Selenium server hosted on the docker container
RSServer$close()
```

5. Selenium Browser Session Component
   The RSelenium Server call **`rSServer()`** will start up the RSelenium Docker container with the **`dockerContainer()`** call shown above. It will also setup the RSelenium server hosted within the Docker container with the **`open()`** method. It also has methods to get status and stop the container.

```
## The outer most component starts all encapusalated components
RSSession <- rSSSession(url="127.0.0.1", portRS=4444, portClient="4444", network="bridge", shm_size="1g",
                        browser_type=test$name, headless=FALSE, image_request=test$image, fresh_pull=FALSE)
## Open a browser instance hosted on a Selenium server within the Docker container
RSSession$open()
## Get status of all the components
RSSession$getStatus()
## Close all components (browser session, server and Docker container)
RSSession$close()
```

5.2 Solution Evaluation

5.2.1 Benefits

The RSelenium Backend performs quite well when working properly.

- The use of the Selenium server and browser session made it easy to ensure only a single browser session was used for each R layoutEngine call.
- The communication protocol used to control the Selenium browser's DOM was abstracted away from our development efforts. Since the Selenium server's primary functionality is offering tight controls of a browser we were able to take full advantage of this. This meant that JavaScript DOM scripting methods can be directly applied to modify its content without the need for worrying about using a communication protocol such at HTTP or a WebSocket.
- Since we can directly modify the contents of the webpage DOM via the Selenium server controls we see live updates in the browser without having to refresh the page. This both improves the user feedback of the using the browser as a debugging device and adds the ability to execute the layout calculation script immediately upon the DOM update.
- Stand alone server seperate from R

5.2.2 Limitations

Some downsides of the RSelenium Backend include the following:

- Scheduling the start-up of the container, Selenium Server and then the web browser session created some difficulties
- Time required to open the full stack was near 5-10 seconds
- Opening the browser application hosted within the docker container in full mode (i.e. no headless) poses difficulties for Windows and MacOS due to the sharing of the host graphics system. It can be finicky to get this working on all host machines easily.
- The browser version is controlled by the Selenium development team which must be supported via one of their docker images. This creates a dependency on this external development team to ensure the latest browsers (and therefore latest web specifications) can be maintained.
- It was found that a latest docker image created a bug with the RSelenium R package that was no longer up to date. Dependencies on both the Selenium docker iamges and RSelenium R Package poses some risk of future bugs or lack of support for the latest features.

5.2.3 Minimum Technical Requirements

- Locally Hosted Server:
  - The Selenium WedDriver has its own integrated server which provides a large set of functionalities. It has an API which can directly modify the contents of the DOM via JavaScript DOM scripting methods [32]. This API negates the need to implement a stand-alone server to serve any content to a URL since the layoutEngine can insert content directly into the `body` of the DOM.
- Browser Integration:
  - The Selenium Docker images come with customized full-featured browsers of the major types: Chromium, Firefox, Opera. In the Backend design, the browser and the Selenium server are run from within the Docker container. This implementation requires the Docker container has access to the host graphics system which can create some dependency and security issues. This is needed so the browser can open on the host machine as a fully operational application. Unfortunately, there isn't strong support for this from

Docker since it is not a primary use-case.

- Bidirectional Communication:
  - The primary strength of this Backend design is the integrated browser controller which allows direct access to the DOM from R. As such, the Backend can perform all types of DOM manipulations such as adding content, executing a JavaScript function and retrieving content. These are the primary communication requirements for the layoutEngine.

5.2.4 Ratings for Target Feature Set

- Cross Platform: **Medium**
  - The primary installation requirements is Docker. This is available on all major operating systems. The Backend solution would require the R users to have Docker installed and the required Selenium Docker images downloaded. This is considered a good solution to provide Cross Platform support however the necessity to run the containerized browser via the host graphics system introduces additional complexity to support all operating systems. At this point, the Backend was successfully run on each OS however there were many difficulties encountered along the way which would indicate that developing a robust solution would be complex.
- Simple Dependencies: **High**
  - As mentioned in the Cross Platform evaluation, the dependencies for this Backend are limited to Docker. This is a good solution to what could otherwise be a very complex installation process. The Selenium team explicitly moved to a Docker containerized solution to reduce the burden of installing and configuring the Selenium WebDriver directly on the host machine. The additional installation requirement is the RSelenium R package which is easily installed from CRAN.
- Industry Support: **Medium**
  - In general, the support for Selenium is very good with updates to their Docker images occurring regularly. The reduction in score was due to the fact the RSelenium R package has to stay in sync with the Selenium updates. It was found that a newly updated Selenium Docker image was incompatible with the RSelenium package which required specifying an older image version. This added complexity and lack of dedicated support for the RSelenium package would pose significant difficulties for a production ready version of the Backend.
- Modern Web Standards: **Medium**
  - In general, the Selenium browsers are kept up to date with the latest released browsers. They are usually only a couple of versions behind the latest available. The deduction comes due to the out-of-sync scenario identified above where only older Selenium images are able to work. This would mean that some of the latest web standards might not be supported.
- Visual Feedback: **High**
  - The Selenium browser is a fully featured browser so there is visual display of the content which is updated live upon any changes to the DOM.

5.3 Summary

In general, the RSelenium Backend offers a simple way of controlling a browser. However, there are several issues that this approach presents to future development. As a result, it does not seem to be a robust solution to continue development with.

6 NodeJS Backend

The NodeJS backend has been developed with the use of three preexisting JavaScript projects. The **webpack DevServer** [33] is used as a local server environment that supports live reload of the browser. The **Express Server** [34] is used as a lightweight web server that is easy to customize. The **WebSocket API** [35] is used to support the bidirectional communication protocol between R and the web browser.

This section first presents the solution design of the layoutEngine Backend using NodeJS. This is then followed by an evaluation of the design based on the Minimum Technical Requirements and Target Feature Set defined earlier in the report.

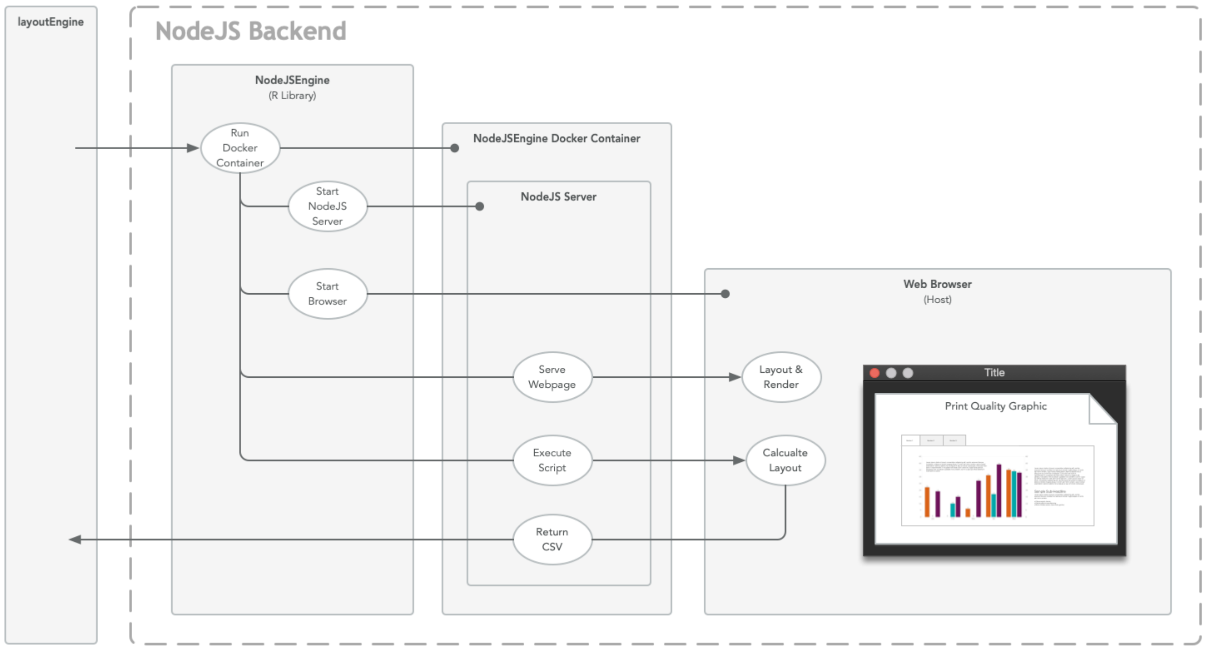
The primary interest in using a NodeJS based Backend was due to the flexibility in which it can be customized to this specific application. The Backend could be built to reside in a number of different locations including a cloud or external server. It could incorporate complex logic and functionality to execute helpful transformations on the server on each side of the data transmission between R and the browser. The motivation behind the NodeJS Backend package includes the following:

- Building the server from scratch would result in very light-weight package
- Easily customize the functionality of it to suit the layoutEngine
- NodeJS is a very popular language with a large community and strong support into the future
- Several Node packages/packages can be used to take advantage of
- Ability to stay at the very fore-front of web language support since we're in control of all depencies
- Few risky dependencies (i.e. RSlenenium and Selenium images)
- Easily work within Docker image for support multiple OS types via communication protocol such as WebSocket or HTTP API
- Can extend functionality of the backend service in an endless variety

6.1 Solution Design

6.1.1 Primary Components

- Webpack Dev Server
- Express Server
- Web Socket
- Docker Container



NodeJS Backend Design

6.1.2 Function Call Relationships

1. RSelenium Backend Arguments
   - url='127.0.0.1'
   - portServer=4444,
   - portClient=4444
   - network='bridge'
   - fresh_pull=FALSE
2. layoutEngineExpress Backend Setup

```
package(layoutEngine)
package(layoutEngineExpress)
options(layoutEngine.backend=ExpressEngine)
```

3. Docker Container Component
   The Selenium Docker container call `dockerContainer()` will start up the Selenium Docker container per the argument settings. It also has methods to get status and stop the container.

```
## User settings for component test
settings <- list(url="0.0.0.0", portServer=8080, portClient=8080,
                 network="host", fresh_pull=FALSE)
## Setup Express Backend Docker container based on user settings
container <- dockerContainer(settings)
## Run the RSelenium docker container
container$run()
## Get Express docker container status
container$getInfo()
## Stop and delete the Express Docker container
```

```
container$close()
```

4. Express Server Component
   The Selenium server call **`rSServer()`** will start up the Selenium Docker container with the **`dockerContainer()`** call shown above. It will also setup the RSelenium server hosted within the Docker container with the **`open()`** method. It also has methods to get status and stop the container.

```
## Setup Express Backend WebSocket server hosted on the docker container
webSocket <- webSocket(settings)
## Get WebSocket server status
webSocket$getStatus()
## Get WebSocket connection status
webSocket$ws$readyState()
## Close Express Backend WebSocket server hosted on the docker container
webSocket$close()
```

5. Express Browser Session Component
   The RSelenium Server call **`rSServer()`** will start up the RSelenium Docker container with the **`dockerContainer()`** call shown above. It will also setup the RSelenium server hosted within the Docker container with the **`open()`** method. It also has methods to get status and stop the container.

```
## The outer most component starts all encapusalated components
WSSession <- WSSession(url="0.0.0.0", portServer=8080, portClient="8080",
                       network="host", fresh_pull=FALSE)
## Open a browser instance hosted on a Selenium server within the Docker container
WSSession$open()
## Get status of all the components
WSSession$getStatus()
## Close all components (browser session, server and Docker container)
WSSession$close()
```

6.2 Solution Evaluation

6.2.1 Benefits

The NodeJS Backend has the following benefits:

- Ability to design server side functionality to extend the usefulness and robustness of the solution
  - For example, transformations or formatting can be performed on the server before sending back to R
  - Very tight compatibiity between a Node server and the browser ensures strong solution design on that side
  - Aim would be to simplify the function calls as much as possible on the R interface
- Live reload still exists via WebPack dev server
- Lightweight dependencies
- Local browser (good and bad)
- Docker container
- Stand alone server separate from R
- Server could reside external to the host if there were ever a need or use-case for

6.2.2 Limitations

Some downsides of the NodeJS Backend include the following:

- Have to implement communication protocol
- Managing multiple browser and/or open tabs needs managing
- Performance hit to stand-up docker container and node server
- Unforeseen issues due to new design with little testing

6.2.3 Minimum Technical Requirements

- Locally Hosted Server:
  - The NodeJS Backend uses a well designed development server to host a web page on the host machine. The server implementation is actually two server implementations that are configured operate together. The webpack development server is used to provide a robust auto reloading of the web page when any changes are detected. The Express serve is used to provide a flexible platform to customize the server with a variety of functionality. This extended functionality is the addition of a purpose built WebSocket connection.
- Browser Integration:
  - The current design relies on the R user to navigate to the correct URL (`http://localhost:port-number`) via their locally installed browser. This provides the flexibility to use any standard web browser. However, it is important to manage the active connections between a R session and a single browser tab. For example, if a user was to open a Firefox and Chrome browser at the same time and navigate to this URL the R session must be explicit in receiving the returning layout calculations from the expected browser.
- Bidirectional Communication:
  - The communication between the R session and the browser is supplied by a simple Web Socket API. The logic of this functionality is split between the server and Backend implementations. The server WS definition manages the connections to ensure messages are routed to the correct clients. The Backend client sends HTML based data to the server when the `grid.html()` function is called via a stringified `JSON` object. The browser client (i) accepts this HTML based data, (ii) loads in into the `body` of the web page DOM, (iii) executes the layout JavaScript function and (iv) accesses the results and sends it back to the Backend client as a CSV within a stringified `JSON` object.

6.2.4 Ratings for Target Feature Set

- Cross Platform: **High**
  - The NodeJS Backend was purposefully designed to be easily used on all major operating systems.
- Simple Dependencies: **High**
  - The Backend requires only Docker and the Backend's Docker image to be available on the host system. All the additional dependencies are installed in the Docker container thereby making the installation relatively straightforward.
- Industry Support: **Medium**
  - All underlying technologies and projects used within the NodeJS Backend are extremely well supported with strong community of developers and applications. There should not be any lack of support and ongoing development for any of these in the foreseeable future. The primary reason for the reduction in score is the fact this Backend implementation is purpose built for the layoutEngine. As such, it would be up to the maintainer of this Backend to ensure ongoing compatibility with any dependencies.
- Modern Web Standards: **High**
  - The R user has access to the latest Web Standards by having the freedom to use the browser of their choice. This is the optimal arrangement.

- Visual Feedback: **High**
  - The local browser can of course offer live visual feedback of the content.

6.3 Summary

7 Comparisons and Recommendations

1. Summary of solution features, benefits and limitations
2. How do they rank with the existing **Backends**?
3. Overview of layoutEngine as a solution to generating print quality graphics
4. Do the new Backends improve its performance?
5. Where should future development work concentrate?

A References

References

[1] Dr. Paul Murrell's of The University of Auckland, Department Web Page. <https://www.stat.auckland.ac.nz/people/pmur002>

[2] Murrell, P. (2018). "Rendering HTML Content in R Graphics" Technical Report 2018-13, Department of Statistics, The University of Auckland. <https://www.stat.auckland.ac.nz/ paul/Reports/HTML/layoutengine/layoutengine.html>

[3] The layoutEngine R Package, GitHub Repository. <https://github.com/pmur002/layoutengine>

[4] The R Project for Statistical Computing, Home Page. <https://www.r-project.org/>

[5] Murrell, P. "R Graphics Third Edition" November 2018, Department of Statistics, The University of Auckland. <https://www.stat.auckland.ac.nz/ paul/RGraphics/chapter1.pdf>

[6] Research Files, GitHub Repository. <https://github.com/kcullimore/layoutengine-research>

[7] Research Files - Docker Container. <https://hub.docker.com/repository/docker/kcull/layoutengine-research>

[8] RSelenium Backend, layoutEngineRSelenium R Package, GitHub Repository. <https://github.com/kcullimore/layoutenginerselenium>

[9] NodeJS Backend, layoutEngineExpress R Package, GitHub Repository. <https://github.com/kcullimore/layoutengineexpress>

[10] NodeJS Backend, Server Code, GitHub Respository. <https://github.com/kcullimore/express-server>

[11] NodeJS Backend, Server Docker Container. <https://hub.docker.com/repository/docker/kcull/layoutengine-express>

[12] CRAN, The Comprehensive R Archive Network, Home Web Page. <https://cran.r-project.org/>

[13] CRAN, Available CRAN Packages By Name, Web Page. <https://cran.r-project.org/web/packages/available.html>

[14] ggtext, R Package. <https://github.com/wilkelab/ggtext>

[15] patchwork, R Package. <https://github.com/thomasp85/patchwork> <https://cloud.r-project.org/web/packages/patchwork/index.html>

[16] grid.layout function within Base R. <https://stat.ethz.ch/R-manual/R-devel/package/grid/html/grid.layout.html>

[17] Murrell, P. R. (1999). Layouts: A Mechanism for Arranging Plots on a Page. Journal of Computational and Graphical Statistics, 8, 121–134. doi: 10.2307/1390924.

[18] layoutEngineDOM GitHub Repository. <https://github.com/pmur002/layoutenginedom>

[19] httpuv, R Package. <https://cran.r-project.org/web/packages/httpuv/index.html>

[20] layoutEnginePhantomJS GitHub Repository. <https://github.com/pmur002/layoutenginephantomjs>

[21] PhantomJS - Scriptable Headless Browser, Home Web Page. <https://phantomjs.org/>

[22] layoutEngineCSSBox GitHub Repository. <https://github.com/pmur002/layoutenginecssbox>

[23] CSSBox Rendering Engine, Home Web Page. <http://cssbox.sourceforge.net/>

[24] W3C Standards, Web Design and Applications. <https://www.w3.org/standards/webdesign/>

[25] The Selenium Browser Automation Project, Home Web Page. <https://www.selenium.dev/documentation/en/>

[26] Docker Container Description. <https://www.docker.com/resources/what-container>

[27] NodeJS, Home Web Page. <https://nodejs.org/en/>

[28] layoutEngine::makeEngine() help page <https://github.com/pmur002/layoutengine/blob/master/man/makeEngine.Rd>

[29] Google, Tools for Web Developers, Puppeteer. <https://developers.google.com/web/tools/puppeteer>

[30] RSelenium R Package. <https://cran.r-project.org/web/packages/RSelenium/>

[31] Selenium Docker Hub. <https://hub.docker.com/u/selenium>

[32] Introduction to the DOM <https://developer.mozilla.org/en-US/docs/Web/API/Document/Introduction>

[33] webpack DevServer. <https://webpack.js.org/configuration/dev-server/>

[34] Express Server Project. <https://expressjs.com/>

[35] WebSocket API <https://developer.mozilla.org/en-US/docs/Web/API/WebSockets$_API$ >

B Appendix

B.0.1 Development Environment

A single Docker container is used to perform research, experimentation, R package development and documentation. This environment was chosen to easily share the development content with others for collaboration and feedback. It will also ensure that any future return to this research can be resurrected with a working code-base independent of software changes.

The report and R development have been performed within Emacs and ESS environment inside of the Docker container. The report is written within the Emacs org-mode markdown language which abstracts some LaTeX syntax while also providing literate programming options which are more flexible than generic markdown or Rmarkdown.

Some basic Docker and Emacs commands are provided to walk the user through some of aspects of the build and editing processes.

1. Docker container description
   Overview: The Docker container is publicly available on Docker Hub with the following image name **kcull\layoutengine-research**. The container is built from the Ubuntu 18.04 image and has R 3.6.3 and Emacs 27.1 installed. The container has been configured to run Emacs in its GUI environment on the host machine.
   User and Home Directory: The user is logged in as a sudo-user with `/home/user/` as the $HOME directory. The sudo password is "password." The working directory is `/project/` which both the shell and Emacs will initialize into.
   Directory Organization: The project also has the primary layoutEngine repositories cloned in the `\opt` directory.
   Directory Hierarchy:

```
# Emacs configuration files
/home/user/.emacs.d/
# Github repository for research paper
/project/
# Github repository for layoutEngine
/opt/layoutengine
# Github repository for layoutEngineDOM
/opt/layoutenginedom
# Experimental code for layoutEngineRSelenium
/opt/layoutenginerselenium
# Experimental code for layoutEngineNodeJS
/opt/layoutenginenodejs
```

2. Host setup and Docker run instructions

The following instructions are provided to recreate the development environment. This has only been tested from within a host machine running Ubuntu 18.04 but is assumed to be compatible with other Debian derivatives.

- GitHub Repository: kcullimore/layoutengine-research
- Docker Image: kcull\layoutengine-research

1 - Download the docker image:

```
$ docker pull kcull/layoutengine-research:latest
```

2 - Create a working directory on the host machine and clone the github repository:

```
$ mkdir /home/$USER/layoutengine-research
$ git clone git@github.com:kcullimore/layoutengine-research.git /home/$USER/layoutengine-research
```

3 - Grant local access to your X server to allow Emacs to run in a local window and the run the docker container (setting is reset upon reboot): **Warning: this exposes your computer. Read more here.**

```
$ xhost +local:
```

4 - Run the docker container:

```
$ docker run --rm -it \
         --network host \
         --privileged=true \
         --env DISPLAY=unix$DISPLAY \
         --volume /tmp/.X11-unix:/tmp/.X11-unix \
         --volume /var/run/docker.sock:/var/run/docker.sock \
         --mount type=bind,source=/home/$USER/layoutegine-research/,target=/project/ \
         --name layoutengine-research \
         kcull/layoutengine-research:latest
```

5 - Once the docker container is up and running verify folder structure has correctly mapped the host directories.

6 - Open Emacs in the container's terminal: `$ Emacs`. The host should launch Emacs in its GUI form (i.e. not within the shell). If this doesn't occur verify steps 4 were followed thoroughly (NOTE: After reboot the display device will have to be provided access again with the `{`$\backslash xhost + local : command$`)`.

7 - From within Emacs perform the following operations to open and recreate the current report

- Opens Treemacs with `M-0`
- Open folder structure to `/project/paper/` with Tab-Enter or Mouse
- Open org-mode markdown file `layoutengine-research-paper.org` with Enter or Mouse double-click
- Make some edits to the file and save with `C-x C-s`
- Launch Export Dispatcher menu with `C-c C-e`
- Create new PDF file with `C-l C-o`

8 - The PDF should have opened automatically which you can scroll through with arrow keys or the mouse scroll wheel. Use `q` key to minimize the PDF buffer.

9 - Close Emacs with `C-x C-c` and exit the container by typing `exit` at the terminal.

10 - Navigate to the project directory on the host machine and verify the new PDF and edited org-mode file were correctly saved.

11 - If the above worked the project appears to be correctly established on the host machine.

3. Emacs within Docker Container

Emacs Terminology

- **buffer:** 'Screen' or 'window' user operates within
- **marking:** Highlighting region of window

Often used commands can be found at `https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf`.

Customized keybindings

- Open emacs configuration file with `C-c e` (Emacs must be restarted for changes)
- Expand all nested/hidden text within *.org file with `Shift-Tab Shift-Tab Shift-Tab`

- Copy, cut and paste with standard keybindings per **Cua Mode**
- Switch visual line wrap with `M-9`
- Switch to truncate long-line view with `M-8`
- Enter/Exit rectangle edit mode with `C-^`
- Enter/Exit multi-edit mode by highlighting word and then `C-u`
- Auto-indent R script (via ESS) by highlighting buffer with `C-x h` and then C-M-}

Document Export

When a PDF version of the document is produced a standard TEX file (*.tex) is also produced after transpilation. This TEX file can be edited and used with a standard LATEX command: `latex report.tex`.

To be continued...

B.0.2 Org-mode examples

1. Font definitions
   Using LaTeX fontspec package [?]
   Sans
   Internet based applications are an increasingly popular way to communicate and interact with complex data.
   Sans italic
   Internet based applications are an increasingly popular way to communicate and interact wtih complex data.
   Sans italic bold
   This might include a business application that assist employees unverstand the current state of the market.
   Serif
   It might also include a news website communicating techincal details from a story such census data.
   Serif italic
   It might also include a news website communicating techincal details from a story such census data.
   Serif italic bold
   It might also include a news website communicating techincal details from a story such census data.
   `Mono type`
   `It might also include a news website communicating techincal details from a story such census data.`
   `Mono Bold type`
   `The quick brown fox 012456789`

2. Sample R code highlighting

```r
##*******10********20********30********40********50********60********70********80
## Problem 2: START => Optical Illusion Example
##*******10********20********30********40********50********60********70********80
## Generate pdf file of plot (capture ends with dev.off() below)
pdf("prob-02.pdf", width = 3, height = 6)
## Create theta values  for each line segments (i.e. 180 degs / 4 = 45 segments)
## Remove elements in the center of vector (i.e. 80-100 degree section)
theta <- seq(0, pi, length = 45)[-(20:26)]
## Set parameters to be used in plot() (R = dummy radius, B = slope of lines)
R <- 1
B <- sin(theta) / cos(theta)
## Setup plot space and define coordinate axes (also remove 'edge buffer')
plot.new()
par(mar = c(0.1, 0.1, 0.1, 0.1))
plot.window(xlim = c(-R, R), ylim = c(-R, R), asp = 1)
## Create the black line segments
for (i in 1:length(B)) abline(a = 0, b = B[i], lwd = 2)
## Create the 2 red vertical lines
abline(v = c(-R/2, R/2), col = "red", lwd = 4)
## Stop image capture
invisible(dev.off())
##*******10********20********30********40********50********60********70********80
## Problem 2: END
##*******10********20********30********40********50********60********70********80
```

3. Sample HTML code highlighting

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1,
            maximum-scale=1.0, user-scalable=0"
    />
    <!-- favicon -->
  </head>
  <body>
    <title>DOM - Testing Application</title>
    <div id="AppDiv" class="app-div"></div>
  </body>
</html>
```

4. Sample CSS code highlighting

```css
.iah-text-Raleway {
  font-family: 'Raleway', sans-serif;
  font-weight: 500;
}

.iah-text-black {
  font-family: 'Roboto Mono', monospace;
  font-weight: 500;
  font-size: 2em;
  overflow-wrap: break-word;
  margin: 10px;
  color: var(--iah-grey-dark);
}
```

5. Sample JavaScript code highlighting

```
var args = []; // Empty array, at first.
for (var i = 0; i < arguments.length; i++) {
    args.push(arguments[i])
} // Now 'args' is an array that holds your arguments.

// ES6 arrow function
var multiply = (x, y) => { return x * y; };

// Or even simpler
var multiply = (x, y) => x * y;
```