Building Big Shiny Apps - A Workflow

Colin Fay

2019-08-09

Contents

Introduction

This book is still at a Work in Progress stage.

Motivation

This book will not **get you started with Shiny**, nor **talk about how to deploy into production and scale your app**. What we'll see is **the process of building the app**. Why? Lots of blog posts and books talk about starting to use shiny or putting apps in production. Very few (if any) talk about this grey area between getting started and pushing into production.

So this is what this book is going to talk about: building Shiny application. We'll focus on the process, the workflow, and the tools we use at ThinkR when building big Shiny Apps.

Hence, if you're starting to read this book, we assume you have a working knowledge of how to build a small application, and want to know how to go one step further.

Book structure

- Part 1 (Building big Shiny Apps) gives a general context about what is a big Shiny Apps and what challenges arise when you're dealing with a big application. It also introduces the `{golem}` package and more general concept about organising your workflow.

- Part 2 to 6 describes a series of steps you can take when you want to build a big Shiny App, and the tooling associated with each.

- Part 7 (Optimizing) tackles the question of optimization, first by presenting some common caveats of big Shiny Apps, and then showing how to optimize R code, and use JavaScript to lighten R work.

About the authors

// TODO

Want to help?

Any feedbacks on the book is very welcome. Feel free to open an issue, or to make a PR if you spot a typo (I'm not a native english speaker, so there might be some waiting to be found ;)).

Other resources

Getting started with Shiny

- Learn Shiny with RStudio
- Getting Started with Shiny
- (WIP) Mastering Shiny
- A gRadual intRoduction to Shiny

Shiny into production

- Shiny in production: Principles, practices, and tools
- Shiny in Production

Suggest a Resource!

Part I

Building Big Shiny Apps

Chapter 1

About Big Shiny Apps

If you're reading this page, chances are you already know what a Shiny App is — a web application that communicates with R, built in R, and working with R. Almost anybody can create a prototype for a small data product in a matter of hours. And no knowledge of HTML, CSS or JavaScript is required, making it really easy to use — you can rapidly create a POC. But what to do now you want to build a big Shiny App?

1.1 A (very) short introduction to Shiny

// TODO

1.2 What's a big Shiny App?

- Well, first, one that includes several thousand lines of code (R and others).
- It's also one that is potentially developed by several coders, working on the same application at the same time.
- It's an application where scaling matters.
- Maintainability and ease of upgrading are important.
- In many cases, Shiny Apps in production are not used by "tech literate" people.
- People rely on this application for making real-world decisions, with real consequences.

Chapter 2

Challenges

2.1 Designing the UI

Choosing a UI is hard — we have a natural tendency, as coders, to be focused on the backend, i.e the algorithmic part of the application. But let's state the truth: no matter how complex and innovative your backend can be, your application is bad if your UI is bad. That's the hard truth. If people can't understand how to use your application, your application doesn't work. No matter how incredible the backend is.

2.1.1 Thinking about accessibility

A user interface needs to be "ready to be consumed" by the broader audience possible. That means people with visual, mobility, or cognitive disabilities.

Web Accessibility is pretty standard concept in the web development world, and you can learn about this straight from the first chapter of learn.freecodecamp.org. -> Screen to speech technology : html semantic tags -> Mobility impairment, people w/ parkinson for example. Exemple of the text input & ENTER button. Making things keyboard friendly. -> ColorBlind, viridis

2.1.2 "Don't make me think"

Two contexts for using a shiny app : professional context or for fun. In both : people want the app to be useable, and easily usable. Work : don't want to lose time figuring out how to use the app. Fun : if you don't understand in a matter of minutes how to use the app, you'll stop using it.

-> KISS principle -> “Don’t make me think”

Try to find a simple, and efficient UI. One that people can understand and use in a matter of seconds. Don’t implement features or visual elements that are not actually needed, just “in case”. And spend time working on that UI, really thinking about what visual elements you are implementing.

-> “Self explanatory”

> All kinds of things on a Web page can make us stop and think unnecessarily. Take names, for example. Typical culprits are cute or clever names, marketing-induced names, company-specific names, and unfamiliar technical names

-> If things are clickable, it has to look clickable

> “As a user, I should never have to devote a millisecond of thought to whether things are clickable—or not”

-> Restrain reactivity

-> Not everything needs to be interactive plots

// TODO: Find resources about simplicity in UI/UX design

2.2 Working as a team: tools & organisation

Big Shiny Apps usually mean that several people will work on the application. For example, at ThinkR, 3 to 4 people usually work on the application. So, how do we organize that?

2.2.1 From the tools point of view:

- Use version control (not sure I have to expand on that topic ;))
- Think of your shiny app as a tree, and divide it as much as possible into little pieces. Then, create one Shiny module by piece. This allows you to split the work, and also to have smaller files — it’s easier to work on 20 files of 200 lines than on one big app.R file.

2.2.2 From the organisational point of view

- Define one person in charge of having the big picture of the app. This person will kick off the project, and write the skeleton of the app, with

the good modules and files structure. This person will also be in charge of accepting new merge requests from other developers, and to orchestrate the master and dev branches.
- List the tasks, and open one issue for each task on your version control system. Each issue will be solved in a separate branch.
- Finally, assign one module to one developer — if it seems that working on one module is a two-person job, divide again into two other submodules. This is a relatively complex task, as the output of one module influences the input of another, so be sure to assign them well.

2.3 Making the app production ready

2.3.1 Scaling

This includes two things: scaling and maintaining. As said in the disclaimer, I won't expand on the topic of scaling, as many have written about that, but here is one piece of advice: make the R process running the app do as little as possible, and in particular prevent it from doing what it's not supposed to do. Which includes: use JavaScript so that the client browser renders things (instead of making R do the work — basic JS is easy to learn), use parallelization and async, and if possible, make the heavy lifting be done outside the R session running the app.

// TODO: link to resources.

2.3.2 Maintaining

Maintainance, on the other hand, is something to think about from the beginning. It includes being able to ensure that the application will work in the long run, and that new features can be easily implemented.

- Working in the long run: separate the code with "business logic" (aka the data manipulation and the algorithm, that can work outside the context of the app) from the code building the application. That way, you can write regression tests for these functions to ensure they are stable.
- Implement new elements: as we are working with modules, it's easy to insert new elements inside the global application.

// TO DO: be more precise on the tooling.

Chapter 3

Structuring your Project

3.1 Shiny App as a Package

In the next chapter you'll be introduced to the `{golem}` package, which is **an opinionated framework for building production-ready Shiny Applications**. This framework starts by creating a package skeleton waiting to be filled.

But, in a world where Shiny Applications are mostly created as a series of files, why bother with a package?

3.1.1 What's in a Shiny App?

OK, so let's ask the question the other way round. Think about your last Shiny which was created as a single-file (`app.R`) or two files app (`ui.R` and `server.R`). You've got these two, and you put them into a folder.

So, let's have a review of **what you'll need next for a robust application**.

First, **metadata.** In other words, the name of the app, the version number (which is crucial to any serious, production-level project), what the application does, who to contact if something goes wrong.

Then, you need to find a way to **handle the dependencies**. Because you know, when you want to push your app into production, you can't have this conversation with IT:

> IT: Hey, I tried to 'source("app.R")' but I've got an error.
>
> R-dev: What's the error?
>
> IT: It says "could not find package 'shiny' ".

> R-dev: Ah yes, you need to install {shiny}. Try to run 'install.packages("shiny")'.
>
> IT: Ok nice. What else?
>
> R-dev: Let me think, try also 'install.packages("DT")'... good? Now try 'install.packages("ggplot2")', and ...
>
> [...]
>
> IT: Ok, now I source the 'app.R', right?
>
> R-dev: Sure!
>
> IT: Ok so it says 'could not find function runApp()'
>
> R-dev: Ah, you've got to do library(shiny) at the beginning of your script. And library(purrr), and library(jsonlite)*.

* Which will lead to a Namespace conflict on the `flatten()` function that can cause you some debugging headache. So, hey, it would be cool if we could have a Shiny app that only imports specific functions from a package, right?

So yes, dependencies matter. You need to handle them, and handle them correctly.

Now, let's say you're building a big app. Something with thousands of lines of code. Handling a one-file or two-file shiny app with that much lines is just a nightmare. So, what to do? Let's split everything into smaller files that we can call!

And finally, we want our app to live long and prosper, which means we need to document it: **each small pieces of code should have a piece of comment** to explain what these specific lines do. The other thing we need for our application to be successful on the long term is tests, so that we are sure we're not introducing any regression.

Oh, and that would be nice if people can get a `tar.gz` and install it on their computer and have access to a local copy of the app!

OK, so let's sum up: we want to build an app. This app needs to have **metadata** and to handle **dependencies** corrrecly, which is what you get from the `DESCRIPTION` + `NAMESPACE` files of the package. Even more practical is the fact that you can do "selective namespace extraction" inside a package, i.e you can say "I want this function from this package". Also, **this app needs to be split up in smaller `.R` files**, which is the way a package is organized. And I don't need to emphasize how **documentation** is a vital part of any package, so we solved this question too here. So is the **testing toolkit**. And of course, the "install everywhere" wish comes to life when a Shiny App is in a package.

3.1.2 The other plus side of Shiny as a Package

3.1.2.1 Testing

Nothing should go to production without being tested. Nothing. Testing production apps is a wide question, and I'll just stick to tests inside a Package here.

Frameworks for package testing are robust and widely documented. So you don't have to put any extra-effort here: just use a canonical testing framework like `{testthat}`. Learning how to use it is not the subject of this chapter, so feel free to refer to the documentation. See also Chapter 5 of "Building a package that lasts".

What should you test?

- First of all, as we've said before, the app should be split between the UI part and the backend (or 'business logic') part. These backend functions are supposed to run without any interactive context, just as plain old functions. So for these ones, **you can do classical tests**. As they are backend functions (so specific to a project), `{golem}` can't provide any helpers for that.
- For the UI part, **remember that any UI function is designed to render an HTML element**. So you can save a file as HTML, and then compare it to a UI object with the `golem::expect_html_equal()`.

```
library(shiny)
ui <- tagList(h1("Hello world!"))
htmltools::save_html(ui, "ui.html")
golem::expect_html_equal(ui, "ui.html")
# Changes
ui <- tagList(h2("Hello world!"))
golem::expect_html_equal(ui, "ui.html")
```

This can for example be useful if you need to test a module. A UI module function returns an HTML tag list, so once your modules are set you can save them and use them inside tests.

```
my_mod_ui <- function(id){
  ns <- NS("id")
  tagList(
    selectInput(ns("this"), "that", choices = LETTERS[1:4])
  )
}
my_mod_ui_test <- tempfile(fileext = "html")
```

```
htmltools::save_html(my_mod_ui("test"), my_mod_ui_test)
# Some time later, and of course saved in the test folder,
# not as a temp file
golem::expect_html_equal(my_mod_ui("test"), my_mod_ui_test)
```

`{golem}` also provides two functions, `expect_shinytag()` and `expect_shinytaglist()`, that test if an objet is of class `"shiny.tag"` or `"shiny.tag.list"`.

- Testing package launch: when launching `golem::use_recommended_tests()`, you'll find a test built on top of `{processx}` that allows to check if the application is launch-able. Here's a short description of what happens:

```
# Standard testthat things
context("launch")

library(processx)

testthat::test_that(
  "app launches",{
    # We're creating a new process that runs the app
    x <- process$new(
      "R",
      c(
        "-e",
        # As we are in the tests/testthat dir, we're moving
        # two steps back before launching the whole package
        # and we try to launch the app
        "setwd('../../'); pkgload::load_all();run_app()"
      )
    )
    # We leave some time for the app to launch
    # Configure this according to your need
    Sys.sleep(5)
    # We check that the app is alive
    expect_true(x$is_alive())
    # We kill it
    x$kill()
  }
)
```

Note: this specific configuration will possibly fail on Continuous integration platform as Gitlab CI or Travis. A workaround is to, inside your CI yml, first install the package with `remotes::install_local()`, and then replace the `setwd (...) run_app()` command with `myuberapp::run_app()`.

For example:

- in `.gitlab-ci.yml`:

```
test:
  stage: test
  script:
  - echo "Running tests"
  - R -e 'remotes::install_local()'
  - R -e 'devtools::check()'
```

- in `test-golem.R`:

```
testthat::test_that(
  "app launches",{
    x <- process$new(
      "R",
      c(
        "-e",
        "datuberapp::run_app()"
      )
    )
    Sys.sleep(5)
    expect_true(x$is_alive())
    x$kill()
  }
)
```

3.1.2.2 Documenting

Documenting packages is a natural thing for any R developer. Any exported function should have its own documentation, hence you are "forced" to document any user facing-function.

Also, building a Shiny App as a package allows you to write standard R documentation:

- A `README` at the root of your package
- `Vignettes` that explain how to use your app
- A `{pkgdown}` that can be used as an external link for your application.

3.1.3 Deploy

3.1.3.1 Local deployment

As your Shiny App is a standard package, it can be built as a `tar.gz`, sent to your colleagues, friends, and family, and even to the CRAN. It can also

be installed in any R-package repository. Then, if you've built your app with `{golem}`, you'll just have to do:

```
library(myuberapp)
run_app()
```

to launch your app.

3.1.3.2 RStudio Connect & Shiny Server

Both these platforms expect a file app configuration, i.e an `app.R` file or `ui.R` / `server.R` files. So how can we integrate this "Shiny App as Package" into Connect or Shiny Server?

- Using an internal package manager like RStudio Package Manager, where the package app is installed, and then you simply have to create an `app.R` with the small piece of code from the section just before.
- Uploading the package package folder to the server. In that scenario, you use the package folder as the app package, and upload the whole thing. Then, write an `app.R` that does:

```
pkgload::load_all()
shiny::shinyApp(ui = app_ui(), server = app_server)
```

And of course, don't forget to add this file in the `.Rbuildignore`!

This is the file you'll get if your run `golem::add_rconnect_file()`.

3.1.3.3 Docker containers

In order to dockerize your app, simply install the package as any other package, and use as a `CMD R -e 'options("shiny.port"=80,shiny.host="0.0.0.0");myuberapp::run_app()'` Of course changing the port to the one you need.

You'll get the Dockerfile you need with `golem::add_dockerfile()`.

3.1.4 Resources

- R packages
- "Building a package that lasts"
- Writing R Extensions
- R package primer - a minimal tutorial

3.2 Using Shiny Modules

// TODO

3.3 Splitting your app into files

// TODO

Chapter 4

Introduction to {golem}

Ok, that's a lot of things to process. Is there a tool that can help us simplify this workflow? Of course there is, and it's called {golem}.

It can be found at https://github.com/ThinkR-open/golem

The stable release can be found on CRAN, and installed with:

```
install.packages("golem")
```

{golem} can only be found on GitHub so you have to install it with:

```
remotes::install_github("Thinkr-open/golem")
```

{golem} is an R package that implements an opinionated framework for building production-ready Shiny apps. It all starts with an RStudio project, which contains a predefined setup for building your app. The idea is that with {golem}, you don't have to focus on the foundation of your app, and can spend your time thinking about what you want to do, not about how to do it. It's built on top of the working process we've developed at ThinkR, and tries to gather in one place the functions and tools we've created for building applications designed for production.

4.1 Getting started with {golem}

Note before using {golem}:

- A {golem} app is contained inside a package, so knowing how to build a package is highly recommended.

- A `{golem}` app works better if you are working with `shiny modules`, so knowing how modules work is heavily recommended.

In the rest of the book, we'll assume you're working in RStudio.

4.2 Understanding `{golem}` package structure

// TODO

Chapter 5

The workflow

-> Overview of the workflow

5.1 Part 1: Design

// TODO

5.2 Part 2: Prototype

// TODO

5.3 Part 3: Build

// TODO

5.4 Part 4: Secure

// TODO

5.5 Part 5: Deploy

// TODO

Part II

Step 1: Designing

Chapter 6

Don't rush into coding

6.1 Designing before coding

Don't rush into coding. I know you want to, because it's what we like to do and what we are good at. But before entering the coding marathon, take time to think about the application and the way it will be deployed and used.

6.2 Ask questions

Take a pen and a piece of paper and draw the app. Talk about it with the people who will use the app, just to decipher what they actually need. Take a moment to talk with the IT. Here are some questions you can ask:

- "Who are the end users of the app?" — This will help you know if the end users are tech literate or not, and what they aim to achieve with the app.
- "How frequently will they use the app?" — The small details of the design & the UI of an app you use on a daily basis is more crucial than when the app is used once a month.
- "What level of complexity and personalization do the users really need?" — People writing app specifications sometimes want more functionalities than what is actually needed by the users.
- "What level of interactivity do you want, and to what extent is it central?" — People love interactive graphs and like when things automatically sync with each other. Yet these two can make the app slower, without any significant gain. For example, being reactive to a `selectInput()` or a `sliderInput()` can lead to too much computation: maybe the user will not succeed to choose the right input the first, second or third time… So

let them do their choice, and add a button so that they can validate when they are ready.
- "How important is it that the app is fast?" — Should you spend a lot of time optimizing the little things?
- etc.

Asking questions, taking notes, and drawing the app help you have a good idea of what is expected and what you have to do now.

So, next step!

Part III

Step 2: Prototype

Chapter 7

Building an "ipsum-app"

7.1 The "UI first" approach

I like to go "UI first". For two main reasons:

- Once the UI is set, there is no "surprise implementation". Once we agree on what elements there are in the app, there is no sudden "oh the app needs to do that now".
- A pre-defined UI allows every person involved in the coding to know which part of the app they are working on. In other words, when you start working on the backend, it's much easier to work on a piece you can visually identify and integrate in a complete app scenario.

So yes, spend time writing a front-end prototype in lorem ipsum. And good news, we've got a tool for you: it's called `{shinipsum}`. The main goal of this package is to create random Shiny elements that can be used to draw a UI, without actually doing any heavy lifting in the backend.

Hence, once you've got a draft of your app on a piece of paper, you can then move to the "ipsum-UI" stage: building the front-end of the app, and filling it with random Shiny elements, with functions like `random_ggplot()` or `random_DT()`.

Another package that can be used to do that is `{fakir}`. This package is designed to create fake data frames, primarily for teaching purposes, but it can also be used for inserting data into a shiny prototype.

Chapter 8

Tools for prototyping

These two tools allow you to prototype a Shiny App and to go "UI first". Learn more:

8.1 Fast prototyping with `{shinipsum}`

// TODO

- https://github.com/ThinkR-open/shinipsum

8.2 Using `{fakir}` for fake data generation

// TODO

- https://github.com/ThinkR-open/fakir

Part IV

Step 3: Build

Chapter 9

Building app with {golem}

Now the UI and the features are set, time to work on the backend.

This part is pretty standard — everybody can now work on the implementation of the functions that process the app inputs, in their own modules. As the UI, functionalities and modules have been defined in the previous steps, everyone (well, in theory) knows what they have to work on.

And also, as said before, there should be no "surprise implementation", as the app has been well defined before.

9.1 Using {golem}

9.1.1 Create a package

Once the package is installed, you can got to File > New Project… in RStudio, and choose "Package for Shiny App Using golem" input.

If you want to do it through command line, you can use:

```
golem::create_shiny_template(path = "path/to/package")
```

This command allows you to create "illegally-named" package (for example, `1234`) by passing the `check_name` argument to `FALSE`. Note that this is not recommended and **should only be done if you know what you are doing**.

Once you've got that, a new RStudio project will be launched. Here is the structure of this project:

```
DESCRIPTION
```

```
¦--dev/
  ¦--01_start.R
  ¦--02_dev.R
  ¦--03_deploy.R
  ¦--run_dev.R
¦--inst/
  ¦--app
    ¦--server.R
    ¦--ui.R
    ¦--www/
      ¦--favicon.ico
¦--man/
    ¦--run_app.Rd
NAMESPACE
myapp.Rproj
¦--R/
  ¦--app_server.R
  ¦--app_ui.R
  ¦--run_app.R
```

If you're already familiar with R packages, most of these files will appear very familiar to you. That's because a `{golem}` app IS a package.

- `DESCRIPTION` & `NAMESPACE`: Package meta-data.
- `dev/`: Scripts that will be used along the process of developing your app.
- `inst/app`: You'll add external dependencies in `www` (images, css, etc). Don't touch `app_ui` and `app_server`.
- `man`: Package doc, to be generated by R.
- `myapp.Rproj`: RStudio project.
- `R/app_server.R`, `R/app_ui.R`: Top level UI and server elements.
- `R/run_app.R`: a function to configure and launch the application.

9.2 dev/01_start.R

Once you've created your project, the first file that opens is `dev/01_start.R`. This file contains a series of commands to run once, at the start of the project.

9.2.1 Fill the DESC

First, fill the DESCRIPTION by adding information about the package that will contain your app:

```
golem::fill_desc(
  pkg_name = "shinyexample", # The Name of the package containing the App
  pkg_title = , # The Title of the package containing the App
  pkg_description = , # The Description of the package containing the App
  author_first_name = , # Your First Name
  author_last_name = ,  # Your Last Name
  author_email = ,      # Your Email
  repo_url = NULL)      # The (optional) URL of the GitHub Repo
```

9.2.2 Set common Files

If you want to use the MIT licence, README, code of conduct, lifecycle badge, and news

```
usethis::use_mit_license(name = "Your Name")  # You can set another licence here
usethis::use_readme_rmd()
usethis::use_code_of_conduct()
usethis::use_lifecycle_badge("Experimental")
usethis::use_news_md()
```

9.2.3 Add a data-raw folder

If you have data in your package

```
usethis::use_data_raw()
```

9.2.4 Init Tests

Create a template for tests:

```
golem::use_recommended_tests()
```

9.2.5 Use Recommended Package

This will add “shiny”, “DT”, “attempt”, “glue”, “htmltools”, and “golem” as a dependecy to your package.

```
golem::use_recommended_dep("")
```

9.2.6 Add various tools

These two functions add a file with various functions that can be used along the process of building your app.

See each file in details for a description of the functions.

```
golem::use_utils_ui()
golem::use_utils_server()
```

9.2.7 If you want to change the default favicon

```
golem::use_favicon( path = "path/to/favicon")
```

You're now set! You've successfully initiated the project and can go to dev/02_dev.R.

```
rstudioapi::navigateToFile("dev/02_dev.R")
```

9.3 Day to Day Dev with {golem}

Now that you're all set with your project init, time to move to development :)

App development should happen through the `dev/02_dev.R` file, which contains common commands for developping.

9.4 Launching the app

To run the app, go to the `dev/run_dev.R` file, and run the all thing.

9.5 dev/02_dev.R

9.5.1 Add modules

The `golem::add_module()` functions creates a module in the `R` folder. The file and the modules will be named after the `name` parameter, by adding `mod_` to the R file, and `mod_*_ui` and `mod_*_server` to the UI and server functions.

```
golem::add_module(name = "my_first_module") # Name of the module
```

The new file will contain:

```
# mod_UI
mod_my_first_module_ui <- function(id){
  ns <- NS(id)
  tagList(

  )
}

mod_my_first_module_server <- function(input, output, session){
  ns <- session$ns
}

## To be copied in the UI
# mod_my_first_module_ui("my_first_module_1")

## To be copied in the server
# callModule(mod_my_first_module_server, "my_first_module_1")
```

In order not to make errors when putting these into your app, the end of the file will contain code that has to be copied and pasted inside your UI and server functions.

9.5.2 Add dependencies

To be called each time you need a new package as a dependency:

```
usethis::use_package("pkg")
```

9.5.3 Add tests

Add more tests to your application:

```
usethis::use_test("app")
```

9.5.4 Add a browser button

Learn more about this: https://rtask.thinkr.fr/blog/a-little-trick-for-debugging-shiny/

```
golem::browser_button()
```

9.5.5 Add external files

These functions create external dependencies (JavaScript and CSS). `add_js_file()` creates a simple JavaScript file, while `add_js_handler()` adds a file with a skeleton for shiny custom handlers.

```
golem::add_js_file("script")
golem::add_js_handler("script")
golem::add_css_file("custom")
```

9.6 Adding these external resources to your app

You can add any external resource (JS, css) into `inst/app/www`.

Then, You'll need to point to these external resources in `golem_add_external_resources()`. For example, if you've created a CSS file with `golem::add_css_file("custom")`, you can add the file with:

```
tags$link(rel="stylesheet", type="text/css", href="www/custom.css")
```

Also, you can list here the use of other packages, for example `useShinyalert()` from the `{shinyalert}` package.

Note: we've chosen to leave it "raw", in the sense that there is a `addResourcePath` and a `tags$head`. If you're comfortable with `{htmltools}`, you can build a `htmltools::htmlDependency`.

9.7 Documentation

9.7.1 Vignette

```
usethis::use_vignette("shinyexample")
devtools::build_vignettes()
```

9.7.2 Code coverage

```
usethis::use_travis()
usethis::use_appveyor()
usethis::use_coverage()
```

9.8 Using {golem} dev functions

There's a series of tools to make your app behave differently whether it's in dev or prod mode. Notably, the `app_prod()` and `app_dev()` function tests for `options( "golem.app.prod")` (or return TRUE if this option doesn't exist).

Setting this options at the beginning of your dev process allows to make your app behave in a specific way when you are in dev mode. For example, printing message to the console with `cat_dev()`.

```
options( "golem.app.prod" = TRUE)
golem::cat_dev("hey\n")
options( "golem.app.prod" = FALSE)
golem::cat_dev("hey\n")
```

```
## hey
```

You can then make any function being "dev-dependant" with the `make_dev()` function:

```
log_dev <- golem::make_dev(log)
log_dev(10)
```

```
## [1] 2.302585
```

```
options( "golem.app.prod" = TRUE)
log_dev(10)
```

Part V

Step 4: Secure

Chapter 10

Build yourself a safe net

Securing your app means two things: testing, and locking the application environment.

10.1 Testing your app

So first, be sure to include tests all along the building process — just like any other R code. As the app is contained in a package, you can use standard testing tools for testing the business logic of your app — as said in the first part, it's important to split the backend functions and algorithm from the user interface. That means that these backend functions can run outside of the application. And yes, if they can run outside of the app, they can be tested the standard way, using {testthat}.

When it comes to testing the front end, you can try the {shinytest} package from RStudio, if you need to be sure there is no visual regression all along the project development. {shinyloadtest}, on the other hand, tests how an application behaves when one, two, three, twenty, one hundred users connect to the app, and gives you a visual report about the connection and response time of each session.

One other tool I like to use is Katalon Studio. It's not R related, and can be used with any kind of web app. How it works is quite simple: it opens your browser where the Shiny app runs, and record everything that happens. Once you stop the recording, you can relaunch the app and it will replay all the events it has recorded. And of course, you can specify your own scenario, define your own events, etc. It's not that straightforward to use, but once you get a good grasp of how it works, it's a very powerful tool.

10.2 A reproducible environment

Secondly, secure your app means that it can be deployed again any time in the future — in other words, you have to ensure you've got a proper handle on the required R version, and of the package versions which are required to run your app. That means that you have to be aware that upgrading a package might break your app — so, provide an environment that can prevent your app from breaking when a package gets updated. For that, there is, of course, Docker, R specific tools like `{packrat}`, or deploying custom CRAN repositories or package manager.

Chapter 11

Secure your work

11.1 Git

Friends don't let friends work on a coding project without version control.

// TO DO

11.2 CI and testing

Testing is central for making your application survive in the long run. The {testthat} package can be used to test the "business logic" side of your app, while the application features can be tested with packages like {shinytest}, or software like Katalon.

// TO DO: more info about the tools + link to resources.

Part VI

Step 5: Deploy

Chapter 12

Send your app to production

Tools for deployment are not the subject of this blog post so I won't talk about this in detail (remember, we are talking about building), but our two tools of choice are Docker & ShinyProxy, and RStudio Connect.

// TODO: link to resources.

Chapter 13

Deploy with {golem}

13.1 Local deployment

// TODO

13.2 Deploying Apps with {golem}

The `dev/03_deploy.R` file contains function for deploying on various plateforms.

13.3 RStudio Environments

// TODO

13.4 Docker

// TODO

Part VII

Optimizing

Chapter 14

Common Application Caveats

14.1 Reactivity anti-patterns

// TODO

14.2 Reading data

// TODO

14.3 R does too much

// TODO

Chapter 15

Optimizing Shiny Code

15.1 Reading data

// TODO

15.2 Caching elements

// TODO

15.3 Keeping things simple

// TODO

Chapter 16

Using JavaScript

16.1 Client-side Optimization

// TODO

16.2 JavaScript <-> Shiny communication

// TODO

16.3 Common JavaScript patterns for Shiny

// TODO

16.4 About `{golem}` js functions

`{golem}` comes with a series of JavaScript functions that you can call from the server. These functions are added by default with `golem::js()` in `app_ui`.

Then they arer called with `session$sendCustomMessage("fonction", "ui_element")`.

This `ui_element` define the UI element to interact with. It can be a full jQuery selector, an id or a class.

16.4.1 golem::js()

- `showid` & `hideid`, `showclass` & `hideclass` show and hide elements through their id or class.

```
session$sendCustomMessage("showid", ns("plot"))
```

- `showhref` & `hidehref` hide and show a link by trying to match the `href` content.

```
session$sendCustomMessage("showhref", "panel2")
```

- `clickon` click on the element. You have to use the full jQuery selector.
- `show` & `hide` show and hide elements, using the full jQuery selector.

16.4.2 About jQuery selectors

- `#plop`: the element with the id `plop`
- `.pouet`: elements of class `pouet`
- `"button:contains('this')"`: buttons with a text containing `'this'`

Note that in html, tags contains attributes. For example:

```
<a href = "https://thinkr.fr" data-value = "panel2">ThinkR</a>
```

contains `href` & `data-value`. You can refer to these attributes with `[]` after the tag name.

- `a[href = "https://thinkr.fr"]`: link with `href` being `https://thinkr.fr`
- `a[data-value="panel2"]`: link with `data-value` being `"panel2"`