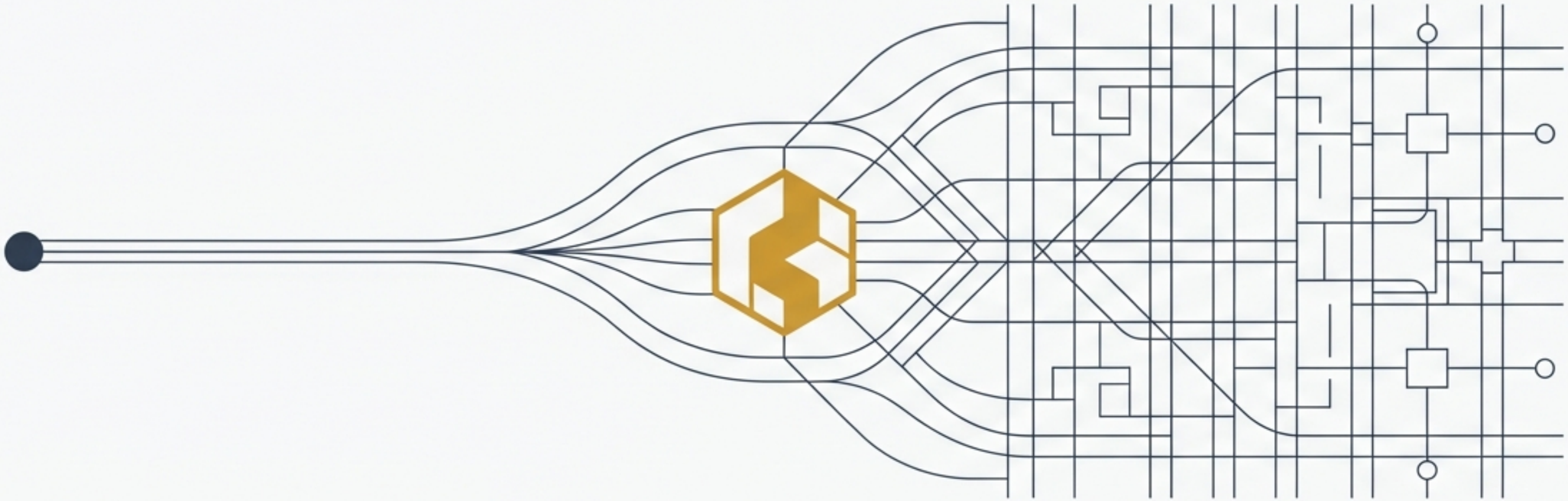
From Prototype to Production: A Beginner's Journey with {golem}

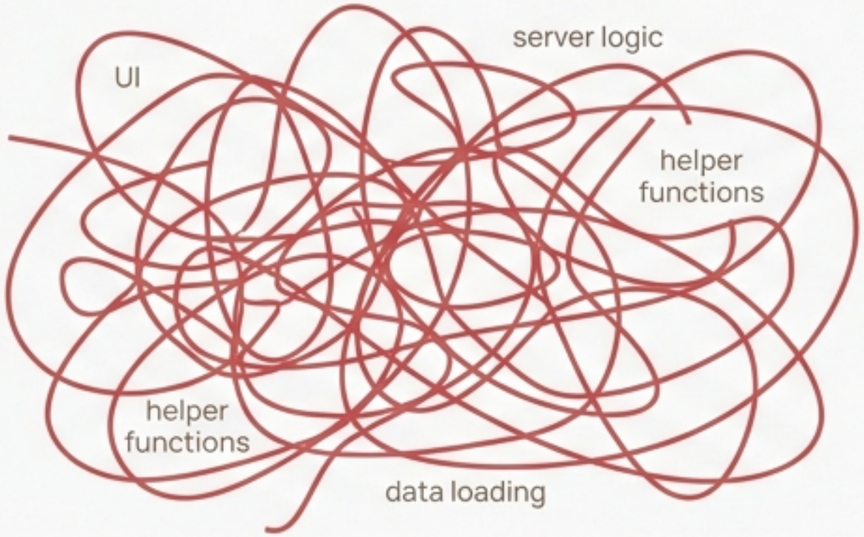
A step-by-step guide to building robust, enterprise-ready Shiny applications.



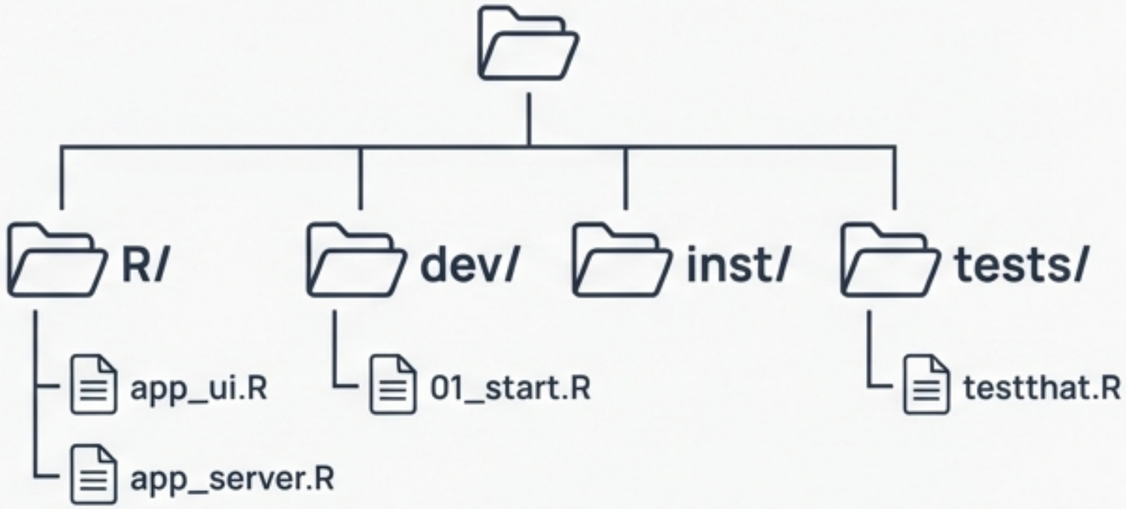
NotebookLM

Your Shiny App Has Grown. Has Your Workflow?

As Shiny applications scale, a single `app.R` file becomes difficult to manage, test, and deploy. This leads to code that is hard to maintain, especially in a team environment. `{golem}` provides a solution by treating your app as a professional software package from day one.



The Prototype Trap.



The Production-Grade Package.

NotebookLM

{golem} is a Development Philosophy, Not Just a Package

`{golem}` is an opinionated framework that enforces best practices. By treating your Shiny app as an R package, you gain access to a mature ecosystem of tools for professional development.

Package-Based Architecture

Enables dependency management, automated testing, and documentation.

Structured Development

Provides a guided workflow (`01_start`, `02_dev`, `03_deploy`) and a standardized project structure.

{golem} Philosophy

Modular by Design

Encourages building with Shiny modules for maintainability and testing.

Production-Focused

Built from the ground up for configuration management, containerization, and deployment.

NotebookLM

Step 1: Initialize Your Project

A single function call creates the entire project structure, complete with pre-configured files and directories for professional development. This is your foundation.

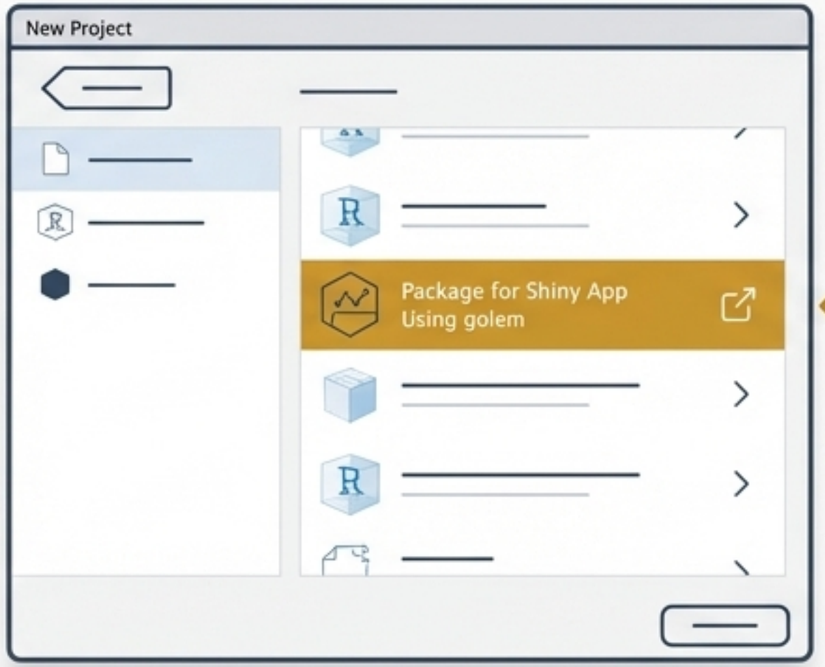
Installation

```
# Install the stable version from CRAN
install.packages('golem')

# Or, the development version from GitHub
# remotes::install_github('Thinkr-open/golem')
```

Project Creation

```
# Create a new golem project
golem::create_golem('my_production_app')
```



Easy setup via RStudio templates.

NotebookLM

Understanding the Project Anatomy

Your new project contains a comprehensive structure. The `dev/` folder guides your workflow, `R/` holds your application logic, and `inst/` stores web assets and configuration.

```
my_production_app/
├── DESCRIPTION
├── dev/
│   ├── 01_start.R
│   ├── 02_dev.R
│   └── 03_deploy.R
├── R/
│   ├── app_server.R
│   ├── app_ui.R
│   └── run_app.R
├── inst/
│   ├── app/www/
│   └── golem-config.yml
└── tests/
    └── testthat/
```

- DESCRIPTION → Package metadata and dependencies
- dev/ → Scripts to guide your entire development workflow
  - 01_start.R ---> Initial project setup & configuration
  - 02_dev.R ---> Day-to-day development workflow
  - 03_deploy.R ---> Deployment configuration scripts
- R/ → Core application logic and modules
- inst/app/www/ → Web assets (CSS, JS, images)
- inst/golem-config.yml → Environment-specific configuration
- tests/testthat/ → Unit tests for your application

NotebookLM

The Workflow Part I: Configure Your Application

The `dev/01_start.R` script walks you through the initial project setup. Use `{golem}` helper functions to fill your `DESCRIPTION` file, set up licensing, and declare recommended dependencies.

```r
# In dev/01_start.R

## Fill the DESCRIPTION ----
golem::fill_desc(
  pkg_name = "myproductionapp",
  pkg_title = "My Production Shiny App",
  author_first_name = "Jane",
  author_last_name = "Developer",
  repo_url = "https://github.com/user/myproductionapp"
)

## Add recommended packages ----
golem::use_recommended_deps()

## Init Testing Infrastructure ----
golem::use_recommended_tests()

## Add specific package dependencies ----
usethis::use_package("plotly")
usethis::use_package("dplyr")
```

Sets up package metadata.

Adds common Shiny dependencies like `{DT}` and `{attempt}`.

Initializes the `{testthat}` framework for unit testing.

Declares a new package dependency for your app.

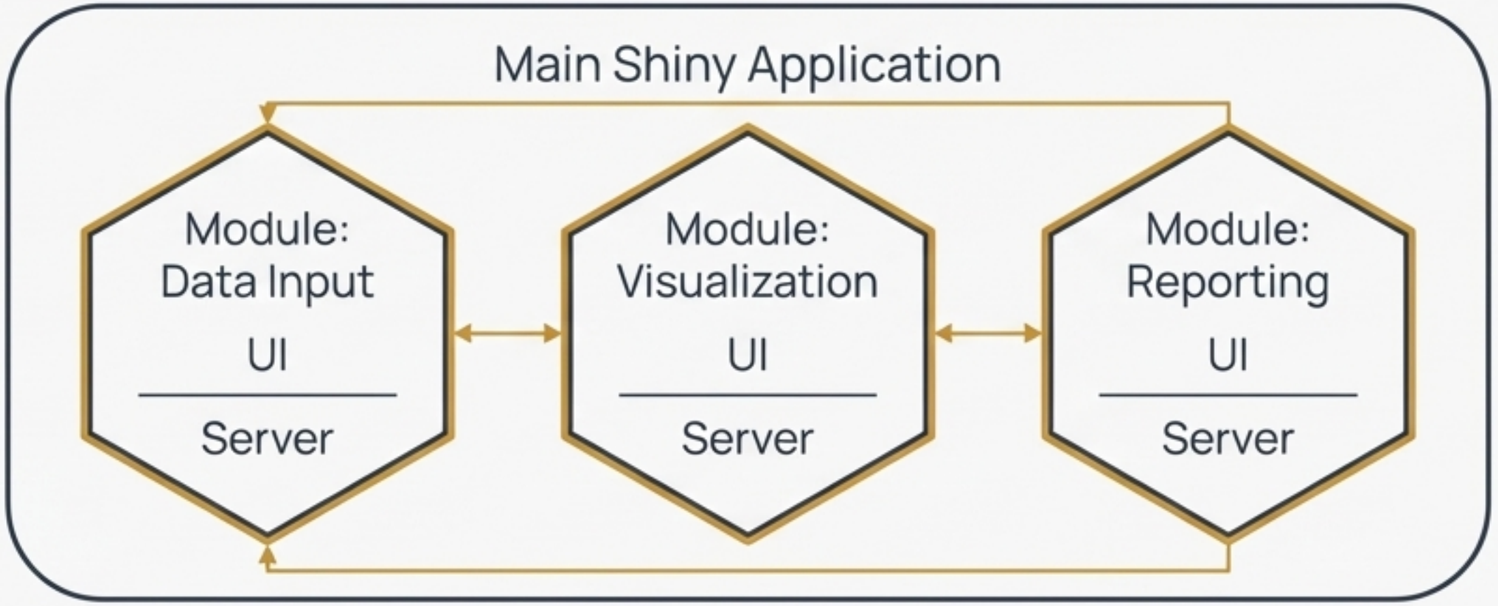
NotebookLM

The Workflow Part II: Build with Modules

{golem} is designed for modular development. Modules encapsulate UI and server logic for a specific piece of functionality, making your app easier to build, debug, and test. We'll build a simple data explorer module for our sample app.

```
# In dev/02_dev.R

## Create a new module with an associated test file ----
golem::add_module(name = "data_explorer", with_test = TRUE)
```



NotebookLM

Implementing the Sample App Module

The `add_module()` command generates `R/mod_data_explorer.R`. This file contains two key functions: `mod_data_explorer_ui()` and `mod_data_explorer_server()`. We'll add a dropdown input and a plot output.

```r
# R/mod_data_explorer.R
mod_data_explorer_ui <- function(id){
  ns <- NS(id)
  tagList(
    selectInput(ns("species"), "Select Species",
                choices = unique(iris$Species)),
    plotOutput(ns("sepal_plot"))
  )
}
```

```r
# R/mod_data_explorer.R
mod_data_explorer_server <- function(id){
  moduleServer(id, function(input, output, session){
    ns <- session$ns
    filtered_data <- reactive({
      iris[iris$Species == input$species, ]
    })
    output$sepal_plot <- renderPlot({
      plot(filtered_data()$Sepal.Length,
           filtered_data()$Sepal.Width)
    })
  })
}
```

The standard server wrapper for Shiny modules.

The `NS()` function creates a unique namespace for module inputs/outputs, preventing ID conflicts.

NotebookLM

Adding Custom Style and Interactivity

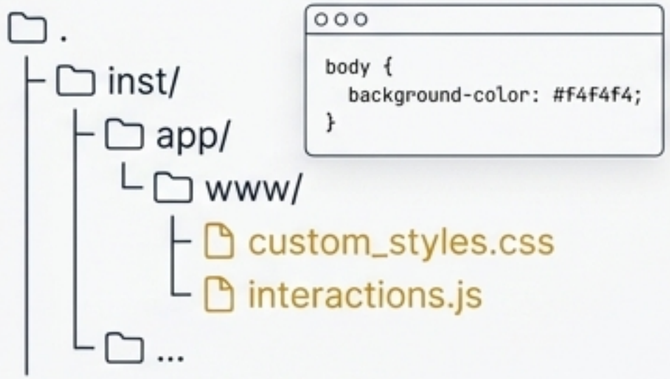
`{golem}` provides a clean workflow for including external JavaScript, CSS, or SASS files. Files are added to `inst/app/www` and are automatically bundled and attached to your application's UI.

1. Create Files

```
# In dev/02_dev.R
golem::add_css_file("custom_styles")
golem::add_js_file("interactions")
```

→

2. Edit Files



→

3. Use JS from R

```
# In your server logic...
observeEvent(input$my_button, {
  # Calls the "alert" function from golem's built-in JS
  golem::invoke_js("alert", "Button was clicked!")
})
```

**Note: `bundle_resources()` in run_app.R handles the attachment automatically.*

NotebookLM

Ensuring Robustness with Unit Tests

Professional applications require testing. Because {golem} builds on the R package structure, you can leverage the powerful {testthat} framework. `golem::add_module(with_test = TRUE)` automatically creates a test file for your module.

```
# In tests/testthat/test-mod_data_explorer.R

test_that("module server logic works", {
  testServer(mod_data_explorer_server, {
    # Set an input value
    session$setInputs(species = "setosa")

    # Check the reactive output
    expect_equal(nrow(filtered_data()), 50)
    expect_true("ggplot" %in% class(output$sepal_plot))
  })
})
```

Tests server logic in isolation, without needing to run the UI.

Simulates user input for testing.

Checks if reactive expressions produce the correct values.

Run all tests with `devtools::test()` or check code coverage with `covr::package_coverage()`.

NotebookLM

Managing Dependencies for Reproducibility

A key benefit of the package structure is explicit dependency management. Use `usethis::use_package()` to declare dependencies in your `DESCRIPTION` file. For full reproducibility, {golem} works perfectly with {renv} to create a project-specific, version-controlled library.

Developer Action

`usethis::use_package("dplyr")`

→

Golem/Package Workflow

Updates `DESCRIPTION` file.

`attachment::att_amend_desc()` can automate this.

+

Renv Workflow

`renv::snapshot()`

→

Result

Creates/updates `renv.lock` file.

Your project now has both a declared dependency list (DESCRIPTION`) and a precise, reproducible environment (`renv.lock`).

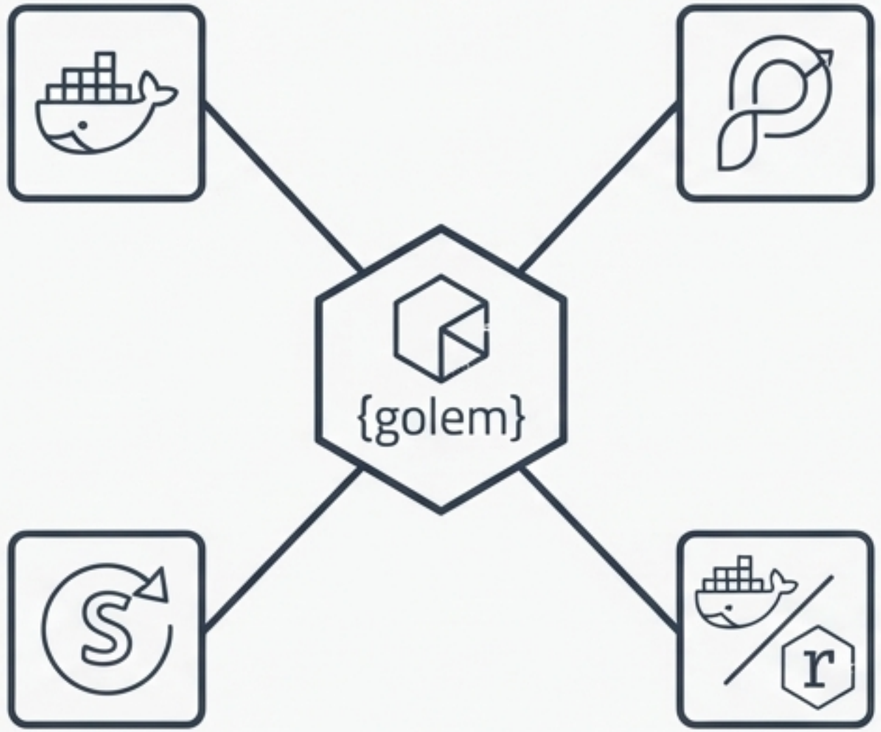
NotebookLM

The Workflow Part III: Prepare for Deployment

The `dev/03_deploy.R` script contains functions to generate deployment-ready configuration files for common platforms like Docker, Posit Connect, and shinyapps.io.

`golem::add_dockerfile()`
-> Generates `Dockerfile`

`golem::add_positconnect_file()`
-> Generates `app.R` entrypoint



`golem::add_shinyappsio_file()`
-> Generates `app.R` entrypoint

`golem::add_dockerfile_with_renv()`
-> Generates a robust, reproducible Docker environment.

NotebookLM

Automating Quality and Deployment with CI/CD

Continuous Integration (CI) and Continuous Deployment (CD) pipelines automate checks and deployments, ensuring high code quality and robustness. By packaging your Shiny app, you can easily integrate with platforms like GitHub Actions.

1. Code Push to GitHub

Developer commits changes.

2. CI Pipeline Triggered (GitHub Actions)

- **Build:** Set up R environment.
- **Test:** Install dependencies (`renv::restore()`), run `R CMD check`, execute unit tests (`devtools::test()`).
- **Quality Gate:** Fail if checks or tests fail.

3. CD Pipeline Triggered (on main branch)

If CI passes, automatically deploy the application to the production server.

NotebookLM

A Practical CI/CD Pipeline with GitHub Actions

You can use `usethis::use_github_action()` to generate a standard R workflow file. This YAML file defines the automated steps that run on every code push, such as checking your package and running tests.

```yaml
# .github/workflows/R-CMD-check.yaml
on: [push]

jobs:
  R-CMD-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4 1
      - uses: r-lib/actions/setup-r@v2 2
      - uses: r-lib/actions/setup-r-dependencies@v2 3
        with:
          packages: any::local_deps()
      - uses: r-lib/actions/check-r-package@v2 4
```

1. Get the code
Points to `actions/checkout@v4`.

2. Set up R
Points to `r-lib/actions/setup-r@v2`.

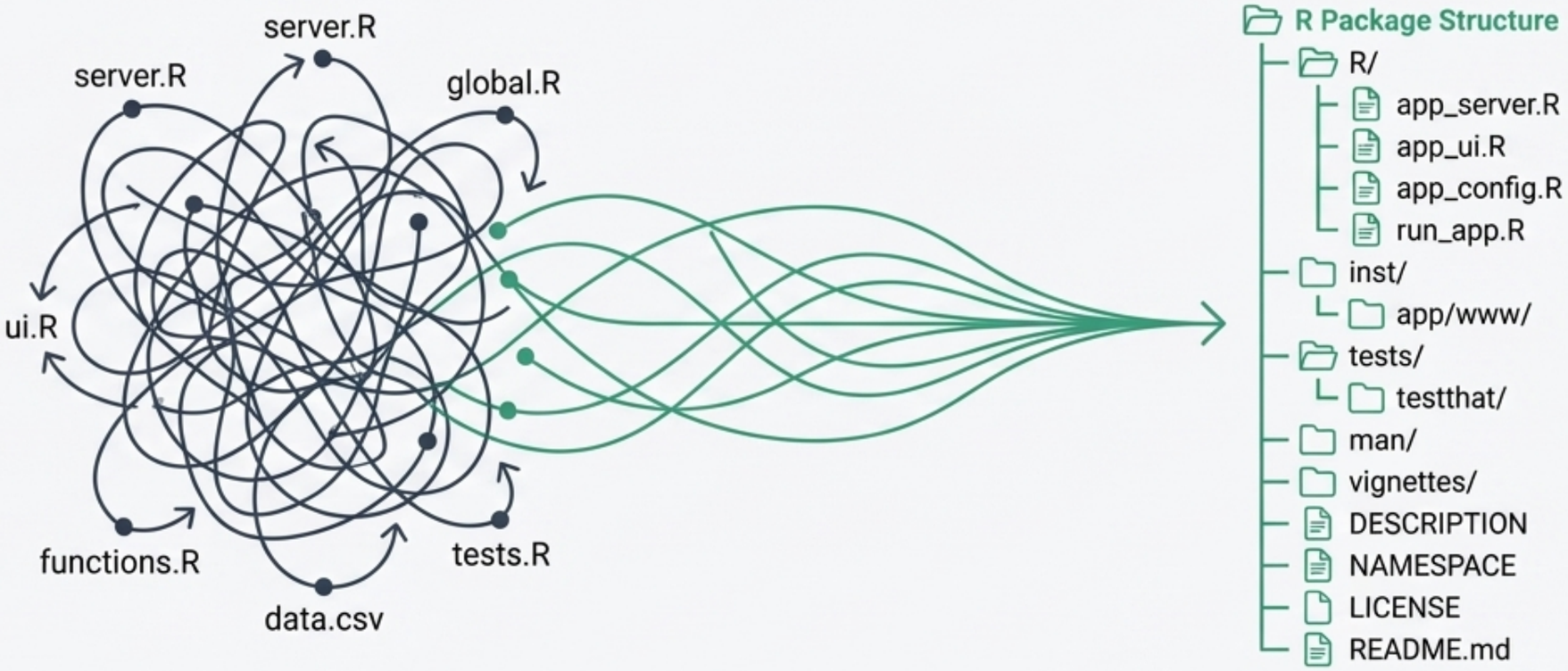
3. Install dependencies
Points to `r-lib/actions/setup-r-dependencies@v2`.

4. Run R CMD check
Points to `r-lib/actions/check-r-package@v2`.

NotebookLM

You're Not Just Building an App; You're Engineering Software.

The {golem} framework transforms how you approach Shiny development. It provides the structure and tooling to move beyond scripting and adopt the robust, scalable practices of software engineering. This journey empowers you to build applications that are not only functional but also maintaiinable, testable, and ready for any production environment.



NotebookLM