Paper OS15

Accelerating Shiny Application Development and Validation with Rhino

André Veríssimo, Appsilon, Lisbon, Portugal
Vedha Viyash, Appsilon, Vellore, India

ABSTRACT

R/Shiny enables the rapid transition from concept to prototype. However, transforming that prototype into a robust, production-ready application presents significant challenges. Frameworks such as Rhino offer structured foundations that simplify this process, providing tools for modularization, dependency management, and extensive testing using {testthat}, {cypress}, and {shinytest2}.
In the pharmaceutical industry, software validation is critical to ensuring compliance with regulatory guidelines. The validation of Shiny applications follows the same fundamental principles as R package validation, incorporating risk-based approaches, test specifications, unit tests, and comprehensive documentation. Unlike R packages, Shiny applications introduce an additional complexity layer due to their interactive nature, requiring validation of both the underlying logic and the user interface.
This paper explores how Rhino facilitates Shiny app development and validation. We outline best practices for structuring and testing Shiny applications, discuss regulatory requirements, and demonstrate how Rhino accelerates the validation process. By leveraging Rhino's built-in features, developers can enhance software quality while maintaining compliance with industry standards.

INTRODUCTION

Shiny applications have become a popular tool for interactive data analysis, particularly in places where the adoption of R is higher. While Shiny enables rapid development, validating these applications pose unique challenges. Regulatory agencies require rigorous testing, documentation, and reproducibility to ensure application integrity. Rhino provides a structured framework for building and validating enterprise-grade Shiny applications. Unlike ad-hoc development approaches, Rhino enforces best practices that can be standardized making them easy for others to review. These standards include modular code organization, automated testing, and dependency management. This paper examines how Rhino simplifies the transition from prototype to validated production application.

COMMON PROBLEM WHILE BUILDING SHINY APPLICATIONS

Rapid development of shiny apps can cause the app to accumulate technical debt during its development, which is a common problem which adds to the long-term cost of making quick, suboptimal development choices that hinder scalability and maintainability. Shiny applications often accumulate technical debt due to:

- Monolithic structures with thousands of lines of code, making onboarding and modifications difficult.
- Limited or missing testing, increasing the risk of undetected errors.
- Hardcoded dependencies that restrict flexibility in deployment.
- Lack of documentation, slowing down development and debugging efforts.

Rhino addresses these issues by enforcing structured development practices, promoting modularization, and integrating testing tools to ensure long-term application health.

STRUCTURING SHINY APPLICATIONS FOR VALIDATION

STAGES OF VALIDATION

Validation of Shiny applications can be broken down into three main stages:

1. **Data Validation** - Ensuring data integrity, consistency, and traceability from source systems (CDASH, SDTM, ADaM).
2. **Package Validation** - Using validated R packages in a controlled environment, leveraging best practices such as source code traceability, metadata storage, and automated reporting.
3. **Application Validation** - Verifying Shiny app functionality, user interactions, security measures, and compliance with regulatory standards.

By segmenting validation efforts, organizations can streamline the process and focus on critical risk areas.

PROJECT MODULARITY WITH RHINO

Rhino encourages a modular architecture, leveraging the {box} package to segment applications into reusable components. Benefits include:

- Clear separation of concerns, making code easier to maintain and scale.
- Improved collaboration, reducing conflicts when multiple developers work on different parts of the application.
- Better debugging capabilities, allowing targeted testing and issue resolution without affecting the entire application.

AUTOMATED TESTING AND CONTINUOUS INTEGRATION

Rhino incorporates multiple layers of testing to ensure application stability:

- Unit testing with {testthat} - Verifies individual functions work as expected.
- End-to-end testing with {shinytest2} and Cypress - Simulates user interactions and detects UI-related issues.
- Automated testing pipelines via GitHub Actions - Ensures new code changes do not introduce regressions.

By integrating these testing frameworks, Rhino enhances reliability while minimizing manual validation efforts.

VALIDATION CHALLENGES AND SOLUTIONS

CHALLENGE: INTERACTIVE NATURE OF SHINY APPLICATIONS

Unlike static R packages, Shiny applications are dynamic and stateful, making validation more complex. The application logic, user interactions, and rendering processes must be validated comprehensively.

Solution: Use Cypress or {shinytest2} for automated UI testing, ensuring consistent application behavior across different environments and user interactions.

CHALLENGE: MANAGING APPLICATION STATE AND DEPENDENCIES

Pharmaceutical applications often require strict control over package versions and system dependencies to maintain reproducibility.

Solution: Rhino integrates with renv, allowing applications to lock dependencies and manage reproducible environments efficiently.

CHALLENGE: REGULATORY COMPLIANCE AND DOCUMENTATION

Regulatory agencies require comprehensive documentation, including test specifications, validation reports, and traceability matrices.

Solution: Rhino's structured approach simplifies documentation generation by enforcing modularization, logging, and automated testing.

RHINO FRAMEWORK OVERVIEW

Rhino is an open-source framework designed to bring software engineering best practices to Shiny applications. It enforces:

- **Modularization** - Breaking applications into reusable modules for better maintainability.
- **Version Control** - Using Git to track changes and ensure collaborative development.
- **Continuous Integration/Continuous Deployment (CI/CD)** - Automating testing and deployment to streamline updates.
- **Security Best Practices** - Implementing authentication, role-based access control, and secure data handling.

PROBLEM WITH NON-STANDARD SHINY APP STRUCTURES

As there are no restrictions to the way we can structure shiny apps, it becomes challenging to onboard new developers to a large shiny app or perform a code review of a shiny app. Just understanding how the app is structured and where the different files are placed becomes a hurdle. This can increase the time it takes to get feedback from FDA when a shiny app is submitted as a supplement.

The GitHub repository: https://github.com/vedhav/shiny_app_structures contains different ways to structure the same shiny app. How the app is structured often times affect the quality of the shiny app significantly. This is why it is recommended to carefully structure the app and setup tools to make the app production ready.

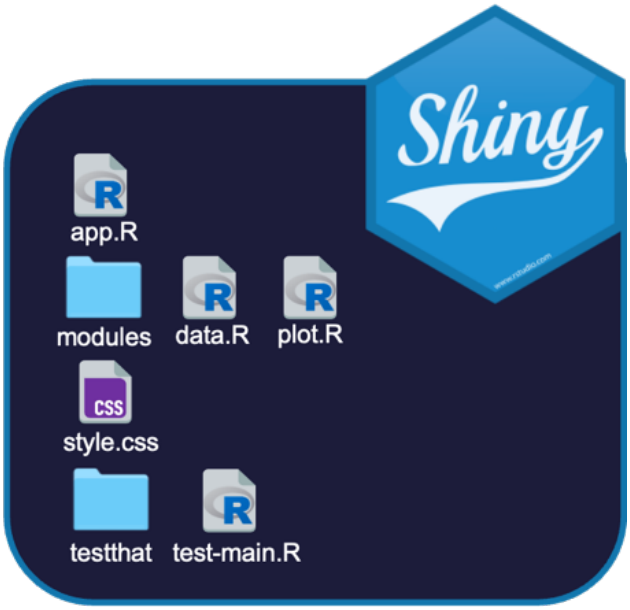
A SINGLE APP.R FILE

This is the most common way of creating shiny apps because of the simplicity. However, once the shiny app gets bigger, it becomes increasingly challenging to maintain it.

SHINY APP THAT HAS CODE DISTRIBUTED IN DIFFERENT FILES

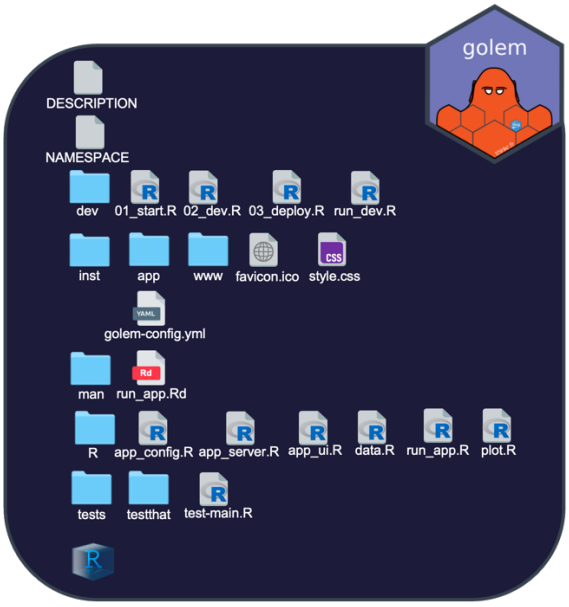
It is a good practice to create dedicated R files that does specific tasks. However, one needs to be careful on how these files are sourced among each other. One common issue with this structure is that it is prone to objects being masked if not managed properly. The R package {box} helps to mitigate this risk by managing how the objects are sourced from different files, it also has the ability to selectively import objects from different R files/packages which increases explicitly and makes it easy to maintain.

SHINY APP THAT HAS CUSTOM STYLE AND UNIT TESTS



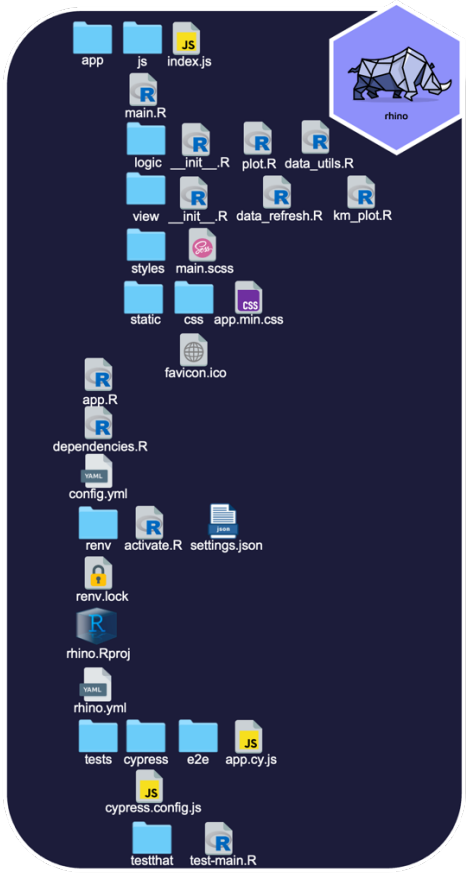
Adding some additional styles using CSS can improve the UI/UX of the shiny app. Implementing unit tests using the {testthat} package helps in identifying issues quicker during development. Note that there is no standard way to implement these tools. The Shiny app developer can choose to use these tools how they would like.

SHINY APP THAT HAS CUSTOM STYLE AND UNIT TESTS



Using a framework like golem helps in introducing some standard into the project. If multiple teams use {golem} to build their shiny apps, they can easily collaborate with each other. Since {golem} creates a shiny app as an R package, it helps in using the established developer practices and tools used to create R package when creating a golem based shiny app.

SHINY APP THAT HAS CUSTOM STYLE AND UNIT TESTS



When creating a shiny app using the {rhino} framework, along with adding a standard structure, it enforces the app developer to follow some good software development practices. This helps in the long term maintainability of the shiny app.

CONCLUSION

Validating Shiny applications is essential for ensuring compliance and maintaining software integrity in regulated industries. Rhino provides a powerful framework that accelerates Shiny application development and validation by incorporating best practices, enforcing modularity, and automating testing.

By leveraging Rhino, organizations can streamline validation workflows, improve software quality, and enhance regulatory compliance. Future advancements in Rhino will continue to support enterprise-grade Shiny applications, making validation more efficient and accessible.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

André Veríssimo
Appsilon
andre.verissimo@appsilon.com

Vedha Viyash
Appsilon
vedha@appsilon.com