Application Shiny : gestionnaire de métadonnées SA_METADATA

Cette application Shiny, codée en R avec **shinydashboard**, est un gestionnaire de métadonnées pour la base SA_METADATA. Elle permet de se connecter à une base PostgreSQL, de créer et mettre à jour des tables de métadonnées (*Test_Info* et *Product_Info_Metadata*), et de détecter les tests et produits dont certaines données sont manquantes. L'interface est organisée en onglets (connexion, scan des tests, scan des produits, saisie manuelle, tables, import/export, debug), chacun offrant des fonctionnalités spécifiques.

Bibliothèques et configuration

Le script charge de nombreuses bibliothèques R : **shiny**, **shinydashboard** (UI), **DT** (tables interactives), **DBI** et **RPostgres** (connexion PostgreSQL), **readxl/writexl** (import/export Excel), **dplyr** (manipulations de données), **shinyWidgets/cssloaders/js** (UI améliorée), **fs/stringr** (fonctions utilitaires). Les valeurs statiques (listes déroulantes pour « GMPS TYPE », « COUNTRY CLIENT », etc.) sont définies en début de code. Les paramètres de connexion à la base sont codés dans DB_CONFIG (hôte, port, user, password, dbname) – la connexion est établie par dbConnect(RPostgres::Postgres(), ...) comme recommandé pour PostgreSQL, et fermée par dbDisconnect. La fonction dbIsValid(con) permet de vérifier la validité de la connexion avant chaque opération.

Création des tables de métadonnées

Deux tables principales sont utilisées :

- **Test_Info** : crée par la fonction create_test_info_table(). Elle contient les informations de tests (source_name, test_name, date, etc.) et des index pour optimiser les requêtes (index sur source_name, test_name, et un index unique sur la paire (source_name, test_name) pour empêcher les doublons).
- **Product_Info_Metadata** : crée par create_product_info_metadata_table(). Elle stocke les métadonnées produits (product_name, code_prod, base, ref, dosage) avec des index analogues sur source_name et product_name.

Chaque fonction de création exécute une requête SQL CREATE TABLE IF NOT EXISTS via dbExecute. Si les tables existent déjà, les fonctions retournent rapidement sans erreur. Cette structure garantit que le schéma de métadonnées est mis en place avant toute lecture ou écriture.

Chargement et sauvegarde des données

Les fonctions load_test_info_from_postgres(con) et load_product_info_metadata_from_postgres(con) lisent les tables existantes en base (dbReadTable(con, "Test_Info"), etc.). Cela utilise **DBI**/**RPostgres** pour récupérer le contenu sous forme de *data.frame* en R. À l'inverse, les sauvegardes s'effectuent soit

par dbWriteTable() (pour insérer de nouvelles lignes) soit par dbExecute("UPDATE ...") (pour mettre à jour). Par exemple, save_test_info_to_postgres() vérifie l'existence d'une entrée identifiée par (source_name, test_name) avec une requête SELECT COUNT(*), puis met à jour ou insère selon le cas. Les fonctions SQL courantes sont utilisées : dbGetQuery pour requêtes SELECT, dbExecute pour UPDATE/CREATE, et dbWriteTable pour inserts. Notamment, dbListTables(con) permet de lister les tables disponibles, et la fonction dbExistsTable(con, "Product_Info") vérifie la présence de la table du script d'analyse. Comme rappelé dans la doc DBI, dbWriteTable() exécute plusieurs commandes SQL pour créer/écraser une table et y insérer les valeurs. De même, dbReadTable(con, "mtcars") peut récupérer le contenu d'une table existante. Ces mécanismes DBI/RPostgres assurent la persistance des métadonnées.

Détection de données manquantes

Deux fonctions parcourent les données importées pour détecter les champs vides :

- **detect_missing_test_info(con)** : lit la table Product_Info générée par le script d'analyse (présumée contenir tous les tests), en extrait les noms de tests, puis compare à Test_Info. Si un test existe dans Product_Info mais qu'il manque dans Test_Info, il est marqué « À compléter ». Si un test est présent dans Test_Info mais certains champs (gmps_type, test_date...) sont vides ou NULL, il est aussi signalé. On utilise ici **dplyr** pour filtrer et manipuler les *data.frame*. Par exemple, filter(is.null(gmps_type) | gmps_type == "") repère les lignes incomplètes. Celles-ci sont affichées dans un tableau interactif.
- **detect_missing_product_info(con)** : analyse directement la table Product_Info. Elle filtre les produits dont des champs (code_prod, base, ref, dosage) sont vides ou NA, en indiquant aussi « À compléter ». Là encore, on applique un filter avec is.null(...) | == "" pour chaque champ.

Chaque fonction retourne un *data.frame* des éléments manquants, réinitialisé pour affichage (valeurs NA converties en chaînes vides). Un message informe du nombre d'éléments détectés, et des exemples sont loggués côté serveur. Ainsi, l'utilisateur peut facilement identifier quels tests ou produits nécessitent une saisie supplémentaire.

Interface utilisateur (UI) – shinydashboard

L'interface utilise **shinydashboard** pour créer un tableau de bord avec un en-tête (dashboardHeader), une barre latérale (dashboardSidebar) et un corps (dashboardBody). La barre latérale définit un sidebarMenu avec plusieurs menuItem, chacun pointant vers un onglet (tabName). Par exemple, l'item **« Scanner Test Info »** correspond au tabName = "scan_tests" et affiche le contenu de tabItem(tabName = "scan_tests", ...). Il est crucial que chaque menuItem et tabItem partagent la même tabName pour que la navigation fonctionnerstudio.github.io. L'ordre des onglets reflète le menu :

- **Connexion SA_METADATA** (onglet connection) : bouton de connexion/déconnexion à la base, affiche l'état (output$connection_status) et les tables disponibles.
- **Scanner Test Info** (scan_tests) : bouton de scan et tableau des tests manquants (via DT::dataTableOutput).
- **Scanner Product Info** (scan_products) : similaire, pour les produits.
- **Saisie Test Info** (manual_test) : formulaire manuel de saisie d'un test. Comprend des selectInput ou textInput pour chaque champ (source_name, test_name, date, etc.), avec validation en temps réel. Deux boutons permettent d'enregistrer ou d'enregistrer+suivant.
- **Saisie Product Info** (manual_product) : formulaire pour renseigner un produit manquant (code_prod, base, ref, dosage).
- **Tables SA_METADATA** (postgres_tables) : affiche les tables Test_Info et Product_Info en base dans deux DT::datatable séparés, avec bouton « Actualiser ».
- **Import/Export** (import) : prévu pour l'import/export Excel (actuellement en développement).
- **Debug** (debug) : affiche des diagnostics (état de connexion, tables existantes, statuts de données).

Les boîtes box() et tabItems() structurant l'UI décrivent les contenus textuels et les sorties. Des alertes colorées (bootstrap) guident l'utilisateur. Le design suit les bonnes pratiques Shiny : on a un sidebarMenu avec des tabName correspondants à des tabItem dans le dashboardBodyrstudio.github.io. Les actionButton déclenchent des observeEvent côté serveur pour réaliser les opérations.

Logique serveur (Server)

La partie serveur définit la logique réactive et les gestionnaires d'événements :

- **Connexion à PostgreSQL** : lors du clic sur « Se Connecter », la fonction create_postgres_connection() ouvre la connexion via dbConnect(RPostgres::Postgres(), ...). Si elle réussit, un reactiveVal postgres_con est mis à jour et l'interface indique la connexion établie (texte vert). De même, le bouton « Se Déconnecter » ferme la connexion (dbDisconnect), remet le statut à faux, et reset l'interface. On stocke le statut de connexion dans connection_status (reactiveVal booléen).
- **Mises à jour des listes dynamiques** : après connexion, on récupère via la base les listes dynamiques nécessaires au formulaire, en exécutant des requêtes (par

ex. get_unique_product_names(con) lit la table Product_Info pour en extraire les noms uniques). Ces listes alimentent les selectizeInput (« Nom Produit », « Source Name ») de manière réactive.

- **Validation des champs** : en temps réel, des observeEvent sur les champs input$test_date, input$sc_request, input$dosage_input appliquent des regex pour vérifier les formats (ex: validate_date_format impose DD/MM/YYYY). Un code CSS (.validation-error, .validation-success) change la bordure des champs et des messages d'alerte sont affichés en cas d'erreur. C'est le principe de feedback immédiat courant en Shiny.
- **Scan des tests et produits manquants** : lorsque l'utilisateur clique sur « Scanner Test Info Manquants », l'observeur correspondant (observeEvent(input$scan_test_info_btn)) appelle detect_missing_test_info(con). Le *data.frame* résultant est stocké dans missing_test_info (reactiveVal) et affiché dans output$missing_test_info_table via DT::renderDataTable. Un statut réactif test_scan_completed permet de conditionner l'affichage du tableau. Un mécanisme similaire gère le scan des produits manquants. Ce pattern (lancer l'opération, remplir une reactiveVal puis rafraîchir l'UI) est typique des applications Shiny.
- **Interactions de double-clic** : dans les tableaux DT des tests/produits manquants, un double-clic sur une ligne déclenche un input$..._cell_clicked contenant l'index de la ligne. Un observeur récupère alors la ligne sélectionnée et pré-remplit le formulaire manuel correspondant (updateSelectizeInput, updateTextInput, etc.). Puis il bascule automatiquement vers l'onglet de saisie (via updateTabItems). Ce mécanisme permet à l'utilisateur de cliquer dans le tableau pour charger un test/produit en mode édition, ce qui améliore l'ergonomie.
- **Sauvegarde des saisies** : les boutons **Enregistrer** du formulaire test ou produit déclenchent un observeur qui collecte les valeurs input$... dans un *data.frame*, les valide (champs obligatoires, formats) puis appelle save_test_info_to_postgres() ou save_product_info_to_postgres(). Ces fonctions effectuent un INSERT ou UPDATE en base. En cas de succès, on notifie l'utilisateur et on relance le scan des éléments manquants pour mettre à jour la liste. Les versions *« Et Suivant »* enchaînent l'enregistrement puis chargent automatiquement le prochain élément manquant (index 1 du tableau).
- **Affichage des tables et diagnostics** : deux DT::renderDataTable affichent le contenu actuel de Test_Info et de Product_Info (cette dernière crée par le script d'analyse). Un bouton « Actualiser » permet de rafraîchir manuellement l'affichage. L'onglet **Debug** affiche en texte brut l'état de la connexion, les tables listées (dbListTables(con)) et quelques statistiques (nombre de lignes dans

chaque table, nombre de tests/prod. manquants). Les reactiveVals missing_test_info() et missing_product_info() sont utilisées pour ces calculs. Enfin, la fonction session$onSessionEnded se charge de fermer proprement la connexion à la base à la fin de la session.

Dans l'ensemble, la logique serveur utilise abondamment **reactiveVal** pour stocker l'état (connexion, liste manquante, statut de scan). Comme l'explique la documentation Shiny, un objet reactiveVal agit comme une variable *réactive* : la lire crée une dépendance, et la modifier notifie les réactifs dépendants. Les autres briques réactives classiques sont utilisées (observeEvent, renderText, renderDataTable, reactive expressions) pour relier les événements UI aux opérations en base et à la mise à jour de l'interface.

Intégration PostgreSQL et opérations DBI

L'application communique avec PostgreSQL via le package **RPostgres** (s'appuyant sur DBI). Le code illustre plusieurs cas typiques :

- **Connexion** : dbConnect(RPostgres::Postgres(), host=..., port=..., user=..., password=..., dbname=...) ouvre une session. Par exemple, on peut écrire con <- dbConnect(RPostgres::Postgres(), dbname = db, host = host_db, port = db_port, user = db_user, password = db_password). La fonction dbIsValid(con) vérifie si la connexion est toujours active avant de l'utiliser.
- **Listage des tables** : dbListTables(con) renvoie tous les noms de tables de la base connectée. Cet appel est utilisé pour afficher les tables disponibles dans l'UI et dans le diagnostic.
- **Lecture d'une table** : dbReadTable(con, "Test_Info") récupère les données sous forme de data.frame. Par exemple, pour afficher la table mtcars temporaire (dans l'exemple RPostgres) on utiliserait dbReadTable(con, "mtcars")rpostgres.r-dbi.org. Dans notre code, cette fonction sert à charger Test_Info existant ou Product_Info.
- **Écriture/MAJ de données** : plusieurs méthodes sont employées. Pour insérer de nouvelles lignes, dbWriteTable(con, "Test_Info", test_data, append=TRUE) écrase pas la table existante mais ajoute (avec row.names=FALSE). D'après la doc, dbWriteTable() crée/écrase la table et y insère toutes les données du *data.frame*. Le code utilise aussi dbExecute(con, SQL, params) pour mettre à jour des lignes existantes (requête UPDATE Test_Info SET … WHERE source_name = $1 AND test_name = $2). Ces appels SQL paramétrés permettent d'éviter les injections et d'actualiser seulement les champs modifiés. Après chaque insertion ou mise à jour réussie, on affiche une notification à l'utilisateur. Les exemples officiels montrent cet usage de DBI : par exemple, on peut exécuter dbGetQuery(con,

"SELECT COUNT(*) FROM Test_Info") pour obtenir le nombre de lignes, ou faire un dbWriteTable(con, "newtable", dataframe) pour importer des données.

- **Vérifications existantes** : avant insertion, une requête SELECT COUNT(*) FROM Test_Info WHERE source_name = … assure l'unicité. De plus, dbExistsTable(con, "Product_Info") renvoie TRUE/FALSE selon que la table existe, ce qui permet de gérer les cas où le script d'analyse génère ou non la table. Les indices uniques (créés au moment des CREATE TABLE via CREATE UNIQUE INDEX) empêchent les doublons à deux niveaux.

En résumé, le code applique les patterns habituels de connexion et manipulation SQL en R : établir la connexion, utiliser dbListTables, dbExistsTable, dbReadTable, dbWriteTable, dbExecute/dbGetQuery, puis refermer la connexion. Cette approche est recommandée pour les applications Shiny en production, où les données sont stockées en base plutôt que dans des CSV locaux.

Organisation et bonnes pratiques Shiny

Enfin, quelques remarques générales :

- **Structure réactive** : L'usage de reactiveVal (par ex. postgres_con <- reactiveVal(NULL)) permet de partager la connexion et d'autres états entre observateurs sans globaliser de variables. Les observeEvent déclenchent des actions uniquement lors d'événements (clics de bouton, changements d'input). Cette programmation événementielle est essentielle dans Shiny : on attend qu'un utilisateur clique sur « Scannez » ou « Enregistrer » pour lancer le code correspondant.
- **Navigation par onglets** : shinydashboard facilite la création d'onglets comme dans l'exemple structurel. Chaque menuItem du sidebarMenu a un tabName qui correspond au tabItem du corps. Dans le code, les fonctions updateTabItems(session, "sidebarMenu", "manual_test") (et similaires) changent dynamiquement l'onglet actif, ce qui permet de guider l'utilisateur vers le formulaire de saisie après un double-clic sur un test manquant.
- **Retour visuel et notifications** : l'interface utilise showNotification() pour informer l'utilisateur des états (succès, erreur, avertissement). Les withSpinner() autour des datatable affichent un chargement animé tant que l'opération est en cours. Les classes CSS personnalisées (.validation-error, .status-connected, etc.) améliorent l'UX en colorant les éléments selon leur statut.

En combinant ces éléments – accès base de données, UI structurée par shinydashboard, logique réactive – l'application propose un outil complet pour gérer les métadonnées, tout en se conformant aux bonnes pratiques de Shiny (séparation UI/serveur, réactivité maîtrisée, encapsulation de la connexion).

Sources : documentation RPostgres/DBI pour la connexion et manipulation SQL datacareer.der postgres.r-dbi.org ; exemples et références Shiny pour reactiveVal et structure shinydashboard rstudio.github.io et shiny.posit.co