DIMSpec Quick Guide - Developing with DIMSpec

Jared M. Ragland*

2023-03-03

Developing with DIMSpec

Working with the project [1] is best done in an Integrated Development Environment (IDE) such as RStudio [2]. This increases transparency and, as all functions are local environment functions, provides a scaffold allowing developers to modify functions for their own needs. It is the hope of these authors that users can leverage what we have built and continue adding to the project to support their specific missions.

As it stands, it was an intentional choice to distribute this project as an R project rather than an R package due to software requirements on user systems. Its complexity and its intended reusability may differ in nature from what you may be accustomed to with R packages. Easily foreseen use cases include installing the project multiple times for different chemical classes, or installing once on a central machine to host apps targeting different databases with the same code set.

Though no universally unique identifier is provided, each installation will be internally consistent and share an overall data structure with other installations to facilitate sharing. Each time a database is created using the associated toolkit it receives a hexed 8-bit random blob (a 16 character alphanumeric string) as a psuedo unique identifier in the `config` table; those wishing to customize an existing installation for the purposes of adding data should use the `make_install_code` function to generate and set this.

Further sections (with the exception of Environment Settings) will assume that the `/R/compliance.R` file has been sourced and environments established. Users are referred to the DIMSpec User Guide for full documentation, including function documentation. At any time from a session where the compliance script has been run, documentation is available using (1) `user_guide` to open the full User Guide in a browser, (2) `fn_guide` to launch a browser pointing to HTML function documentation index, or (3) `fn_help(X)` where `X` is the name of a function to see documentation for that function.

*NIST | MML | CSD | jared.ragland@nist.gov

[1] NIST-developed software is provided by NIST as a public service. You may use, copy, and distribute copies of the software in any medium, provided that you keep intact this entire notice. You may improve, modify, and create derivative works of the software or any portion of the software, and you may copy and distribute such modifications or works. Modified works should carry a notice stating that you changed the software and should note the date and nature of any such change. Please explicitly acknowledge the National Institute of Standards and Technology as the source of the software.

NIST-developed software is expressly provided "AS IS." NIST MAKES NO WARRANTY OF ANY KIND, EXPRESS, IMPLIED, IN FACT, OR ARISING BY OPERATION OF LAW, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND DATA ACCURACY. NIST NEITHER REPRESENTS NOR WARRANTS THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT ANY DEFECTS WILL BE CORRECTED. NIST DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE SOFTWARE OR THE RESULTS THEREOF, INCLUDING BUT NOT LIMITED TO THE CORRECTNESS, ACCURACY, RELIABILITY, OR USEFULNESS OF THE SOFTWARE.

You are solely responsible for determining the appropriateness of using and distributing the software and you assume all risks associated with its use, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and the unavailability or interruption of operation. This software is not intended to be used in any situation where a failure could cause risk of injury or damage to property. The software developed by NIST employees is not subject to copyright protection within the United States.

[2] Any mention of commercial products is for information only; it does not imply recommendation or endorsement by NIST.

Environment Settings

Many settings for the project are controlled by environment establishment scripts. This is intended to isolate the session for a project to a particular use case (i.e. focus on a single database) and make using project functions much more convenient as function parameters will in most cases default to session settings. Settings are largely specific to particular aspects (e.g. logging settings are determined in large part in the logger environment file). These settings may be found at:

1. `/config/env_glob.txt` controls mainly which aspects of the project are activated by default, as well as system information (e.g. PATH aliases), and information about the database itself (e.g. title, name, etc);
2. `/config/env_R.R` controls aspects specific to R (or that require R functions to generate) and controls aspects which packages are used for database communication, and which files are sourced, among others;
3. `/config/env_logger.R` controls the basic settings for project logging, which are used to write log files to the directory, define namespaces, and define log interaction functions;
4. `/inst/apps/env_shiny.R` controls aspects related to enabling shiny apps for this session, as well as launching the API server if required by an application;
5. `/inst/plumber/env_plumb.R` controls aspects related to the plumber API service including the session object name for the service watcher, which files are made available to it, and the logger settings specific to the plumber instance;
6. `/inst/rdkit/env_py.R` controls aspects specific to the python integration and may be highly system dependent; if you are accustomed to working with python environments or have a non-standard install, pay close attention to settings in this file.

Once these files are sourced into an R session, their package requirements will be loaded and a series of logical scalars will be present with names like `RENV_ESTABLISHED` indicating which have been activated and all settings will be present in the session. Several packages are required for DIMSpec operation and will be installed if necessary during the environment resolution stage. For new installations this may take a considerable amount of time and may result in package version conflicts; user attention to feedback during package installation is advised. See more detail about each of these files and their settings in the "Project Set Up" section of the DIMSpec User Guide. Some of these settings will be discussed in subsequent sections.

DIMSpec Databases

This section describes in brief some aspects of databases produced by the DIMSpec project; more details are available in the DIMSpec User Guide.

DIMSpec databases are SQLite files and as such can be used by any software able to open and interact with them. By default they are constructed in the write-ahead logging (WAL) mode to improve speed and provide better concurrency. It is anticipated that most interactions will be read only. Network access is provided by the plumber API as WAL mode does not work across network connections. Similarly, all connections made should set foreign key enforcement (e.g. `"pragma foreign_keys=on"`) when connecting as this is not the default when opening SQLite databases; this is done automatically using the provided `manage_connection` project function and is the recommended method of connecting your R session to the database.

A major goal of the project was to support rapid development of portable databases for mass spectrometric non-targeted analysis. Toward the end, creation of new databases using the DIMSpec schema has been simplified as much as possible. Importing data is an exercise left to users of the project as project inputs may vary widely, but example import routines are based around the `mzML` output format from msConvert

(Adusumilli, Ravali and Mallick, Parag 2017) with annotations provided by using the NIST Non-Targeted Analysis Method Reporting Tool.

The project ships with a populated database of perfluoroalkyl substances. To connect to the database defined in the environment file, use `manage_connection`; this can also be used to flush the write ahead log and reconnect at any time provided another connection does not already exist. To create a new container database, use the `build_db` function. The default title and file name are defined in the `env_glob.txt` file; to build a new database, provide the function with a different name or set `archive = TRUE`. If the defaults are used, the current database can be easily overwritten and constructed again from scratch. If using `build_db` to build a new database, provide in `env_R.R` which scripts in standard query language (SQL) to use to build and populate as the names of files (typically in the `config` directory). If `sqlite` is available on your system, R will use that preferentially to read those SQL files; this is the recommended method for constructing DIMSpec databases. For example, the following function call will create a new database as file `test_db.sqlite` in the project directory without populating it, and create a corresponding connection object in the session named `con2`.

```
build_db(
  db = "test_db.sqlite",
  populate = FALSE,
  db_conn_name = "con2",
  connect = TRUE
)
```

DIMSpec uses an opinionated comment structure in the SQL entity definitions to provide more information about each table and column in the schema; this is in contrast to various SQL management systems which will hold and parse comments or descriptions in various ways. Opinionated comments are used to build both a data dictionary and an entity-relationship map from R functions included with the project to better document the database schema and allow R to interact with it in programmatic ways.

The schema is organized into a series of "nodes"; by default each node is constructed from a single `.sql` file in the `/config/sql_nodes` directory. See the Database Schema section of the DIMSpec User Guide for more detail about the schema. The current data dictionary is written to the project directory as a JSON file with the suffix "_data_dictionary.json"; the first part of this file name will be determined by environment settings as the database name, its version, and its build date. This is read into an active session as the `db_dict` list object. Similarly, the entity-relationship map is produced by the `er_map` function and is available to an active session as the `db_map` list object. These objects are your in-session guide to understanding the database schema. A full size entity-relationship diagram is also available in the project directory as `ERD.png`.

Included for many normalized tables is a corresponding denormalized view (e.g. `compounds` and `view_compounds`). If a table includes foreign key relationships and your application needs the "value" instead of the "index" for those columns, load the view instead. Such views are automatically created through the `sqlite_auto_view` function by reading the output of the `er_map` function (typically available to your R session as the `db_map` object) and generating the SQL commands defining the denormalized view. SQLite databases by default do not enforce foreign key relationships. While this is turned on when an R session connects to the database, automatic triggers were created from scripts using the `sqlite_auto_trigger` function to enforce referential integrity in cases where the connection may be established by other mechanisms; these do not enforce integrity (e.g. no text in integer columns) *per se* they do instead insert unknown values into the corresponding normalization table and update the foreign key relationship appropriately. To include or exclude these scripted additions, edit the SQL file used to construct the database (typically at `/config/build.sql` by removing references to `auto_triggers.sql` and `auto_views.sql`).

R

DIMSpec was developed largely prior to R v4.0 but was updated to reflect changes and finish development in R v4.1. The DIMSpec project contains many package dependencies which are loaded during sourcing of the compliance script or environment resolution. Loading your favorite packages is encouraged, but to minimize conflict potential and for stability reasons this should be done after loading any packages on which DIMSpec relies (i.e. after the compliance script has executed). Packages required by DIMSpec are not renv-locked or specified by package version at this time. See the References section of the DIMSpec User Guide for package versions used for this release.

Many convenience functions exist for advanced work in the database through R, powered by the `DBI` and `RSQLite` packages. The DIMSpec project is fully supported by the "tidyverse" methods of interacting with database connections and will allow for as many connections as necessary, but most functions defined herein rely on the database defined in the `env_glob.txt` file for function argument default. A pair of convenience functions for development are provided in the `/R/app_functions_devonly.R` file: `open_proj_file` will open any file by name regardless of its location in the project directory and will prompt you if multiple files match the provided string; `open_env` is a shortcut specific to environment definition files which can be fed a short hand reference (such as "R", "global", etc.) to quickly access those settings.

The project directory follows many conventions familiar to packages, with the majority of R code in the `/R` directory, help files in the `/man` directory, vignettes in the `/vignettes` directory, and shiny apps, plumber API, and rdkit references in the `/inst` directory. The long form pre-built User Guide is located in the `/dimspec_user_guide` directory; access this in your system browser using the `user_guide` function from an active DIMSpec session.

Several convenience functions are provided to improve the development experience once the compliance script has been executed.

1. `start_api` will spin up the plumber API integration if the session was started with `USE_API` set to `FALSE` [3].
2. `api_reload` will start or restart the API service similarly to `start_api` with a few more convenience parameters (e.g. `background = TRUE` to launch it as a background service).
3. `start_rdkit` will enable python integration and provide chemometrics capabilities via `rdkit` if the session was started with `INFORMATICS` or `USE_RDKIT` set to `FALSE` [4].
4. `start_app` will launch a shiny application by name; options are contained in the `SHINY_APPS` session object if `USE_SHINY` was set to `TRUE` when the session started.
5. `user_guide` will launch a web browser to the DIMSpec User Guide from the console (either hosted or local).
6. `fn_guide` will launch a web browser pointing to the project function help documentation.
7. `fn_help` will open the documentation for any given DIMSpec R function (either in the RStudio Help pane or in a web browser).

This is far from an exhaustive list, but rather those that may be of use during development as certain aspects of the project are activated or deactivated during the compliance script.

[3] There may be conflicts with `reticulate` if packages that use the Java Development Kit (e.g. rcdk) are loaded. At this point these are unresolvable and it is recommended that developers stick with either `rdkit` or `rcdk` as their needs dictate. Setting `INFORMATICS` to `TRUE` and `USE_RDKIT` to `FALSE` will activate `rcdk`, but functions using it are not supported in this release.

[4] For users unfamiliar with `reticulate`, once a python environment has been bound to an R session, it cannot currently be unbound or reassigned. To switch or deactivate python environments, your R session will need to be terminated and reloaded. Exercise care to maintain your global environment when doing so, as this may require a complete shut down and reload of RStudio.

Database Communications

Functions dealing with database manipulation and communication are provided in the `/R/db_comm.R` and `/R/api_generator.R` files. The most commonly used functions during development are likely to include:

1. `manage_connection` is the default way to (re-/dis-)connect database connections, and will automatically clean up after itself if `rm_objects = TRUE`; it can also be used with `reconnect = FALSE` to clean up any hanging connections and force a WAL flush;
2. `close_up_shop` will automatically clean out all database connected objects in your session and tidy up after itself, including database connections, promises, and API services running in the background and, if `back_up_connected_tbls = TRUE` will maintain as `collect`ed tibbles any promises that currently exist;
3. `pragma_table_def` and `pragma_table_info` will reveal properties of a database entity by querying the database connection directly, and can include SQL comments;
4. `data_dictionary` uses the above to create an indexed single object LIST expression containing information about the database schema and is handy for querying properties and, perhaps more usefully, comments (run the adjacent `save_data_dictionary` function to save the output to disk and update the cached version);
5. `er_map` (similarly to the `data_dictionary` function) uses the above to parse the SQL build definitions and create mappings between tables that are then available to the R session, allowing for programmatic querying of the database structure for FK relationships and normalization value search interactions (this map is used heavily to simplify e.g. import processes);
6. `active_connection` will check that a connection is still valid and return a boolean;
7. `check_for_value` checks whether a value exists (allowing for case sensitivity and fuzzy checks) in the distinct values of a given column and can be serialized (e.g. `lapply`) to check multiple columns;
8. `resolve_normalization_value` is a workflow logic wrapper to check for and resolve (adding if necessary) a value in a normalization table and returns the associated PK for inclusion in a source table and can be run in case (in)sensitive or fuzzy match modes, and in the case of multiple matches during interactive use the user will be prompted at the console for a resolution;
9. `dataframe_match` is a data.frame centered search function to build a database query searching for matching records, rather than having to issue or build such; and
10. `build_db_action` allows construction of the SQL for more complicated queries; while tidyverse style queries often work, there are times where development will require much more complicated interactions. This function:
    - ameliorates SQL injection by using SQL interpolation functions from the DBI package;
    - uses `clause_where` to construct database queries from common R object structures like data.frames and lists;
    - wraps the logic for `SELECT`, `INSERT`, `UPDATE`, and `DELETE` transactions (including a verification step prior to executing `DELETE` operations);
    - provides `NROW` and `GET_ID` query analogs to simplify common data.frame like queries for convenience (`GET_ID` is similar in many respects to a simplified case `check_for_value` that assumes presence and known column); and
    - powers a majority of database interactivity within the project, especially so for the plumber API.

This is not an exhaustive list, but rather those that the developers of DIMSpec most often leverage while working with attached databases to develop new database transactions and workflows.

Argument Verification

Often developing on top of new packages in R can cause opaque error messages and overly relies on early stops if assumptions for values provided to function arguments are violated (or the function simply fails). The DIMSpec project includes an argument verification function (`verify_args`) that is called as part of most functions. It accepts a list of arguments and a list of conditions to check against and provides verbose feedback on verification and failure and, more transparently, the mode and reason for failure. The easiest way to include such argument verification in developed functions is to add `stopifnot(arg_check(as.list(environment()), list(A1 ... Ai))$valid)` where `A1 ... Ai` is a series of lists describing the verification checks with names matching all provided arguments. For example, this check in `pragma_table_def` includes:

```
arg_check <- verify_args(
  args       = as.list(environment()),
  conditions = list(
    db_table = list(c("mode", "character"), c("n>=", 1)),
    db_conn  = list(c("length", 1)),
    get_sql  = list(c("mode", "logical"), c("length", 1)),
    pretty   = list(c("mode", "logical"), c("length", 1))
  ),
  from_fn    = "pragma_table_def"
)
stopifnot(arg_check$valid)
```

See `verify_args` in the function documentation (or call `fn_help(verify_args)` from the console) for more details. This provides a more robust development experience but does carry some level of computational overhead. For this reason, argument verification can be turned off by setting `VERIFY_ARGUMENTS` to `FALSE` or `MINIMIZE` to `TRUE` in the `env_R.R` environment settings.

Logger

Most functions within DIMSpec include some level of logging to facilitate debugging and development. Logging provides a tremendous amount of support and maintenance potential, but does carry some level of computational overhead. For this reason, logging can be turned off by setting `LOGGING_ON` to `FALSE` in the `env_glob.txt` environment settings or `MINIMIZE` to `TRUE` in the `env_R.R` environment settings. While logging functionality in DIMSpec is largely based off the `logger` package, it will function for console printing without the package installed if, for some reason, it is not available. Control the basic settings for logging within the DIMSpec project with the `/config/env_logger.R` file. This file allows for customization, but care should be used with certain aspects. By default, the `LOG_DIRECTORY` where logs will be written is `/logs` in the project directory; DIMSpec does not ship with logs from other installations and will spin up logging specific to your installation.

Logging settings are stored in the `LOGGING` session object, a nested list of settings for each namespace containing the following properties:

1. `log` is a boolean setting of whether or not to record logs for namespace `ns`;
2. `ns` is a character scalar of the namespace to use while logging, typically matching the name of a `LOGGING` element;
3. `to` is a character scalar of where logs should be recorded, must be one of `file`, `console`, or `both`;
4. `file` is a file path character scalar which determines the file within `LOG_DIRECTORY` associated with the namespace where logs will be written if `to` is one of `file` or `both`; and
5. `threshold` corresponds to the ranked factor list of log levels below which to ignore (e.g. `TRACE` < `DEBUG` < `INFO` < `SUCCESS` < `WARN` < `ERROR` < FATAL) and correspond to the Apache logj4 log levels.

The environment file here also defines a trio of functions to update the logger settings once changed in a session, and view logs either printed to the console or as a data.frame object for search/filtration/etc.

To issue log messages, use the `log_it` function.

Its basic syntax is to set the logging level, the message, and the namespace to which to write the log (with the default being "global"). It is often useful to check the setting of `LOGGING_ON` prior to (e.g. `if(LOGGING_ON) log_it("info", "example", "global"))`) but this is not strictly required. Having the log level be in the list of Apache levels is also not required, though it is for the `logger` package to produce colorful print messages. Another common use case for debugging is to call the wrapper function `log_fn` which will by default produce a trace level log message at the default namespace for the function from which it was called; typically this is called once at the start (`log_fn('start')`) and once at the end (`log_fn('end')`) of a function to enable a full call stack trace log. As with all `log_it` calls, calls to the namespace must have a `threshold` of `trace` or these will be ignored.

Calling `log_it` with a namespace that has not yet been established will conveniently create a new logging namespace if `add_unknown_ns = TRUE`; this can be leveraged to write a completely new namespace for logging at any time. Currently, logging is strictly enforced for shiny apps and the plumber API and will be turned on for those if launching directly from the console or a script (e.g. `shiny::runApp("path_to_app")` or spinning up a background API process with `api_reload(background = TRUE)`). This is useful for debugging and maintenance purposes. For example, the plumber API includes a filter endpoint at the start of its response chain that will log the location from which all requests are received.

It may be useful at times to flush the logs directory. Use the `flush_dir` function with the logs directory (e.g. `flush_dir(LOG_DIRECTORY)`) to flush the log files. Some additional details are availble in the Technical Details > Logger section of the DIMSpec User Guide.

A logging table is provided in the database schema but currently is not used for transaction logging by DIMSpec functions as distributed. It is recommended that developers take advantage of this (perhaps by triggers or code-driven database executions) based on workflow needs, but in no way is required.

Python

To include chemometric functionality from the excellent RDKit package, reticulate is used to integrate a python environment into DIMSpec projects. Most information is available in the DIMSpec User Guide, but the most relevant settings for DIMSpec developers include references to the python environment and how to establish it on any given system. In most cases, ensuring that `PYENV_NAME` is either a valid environment for running a reticulated version of rdkit OR does not exist, and that `CONDA_PATH` refers to the system PATH alias or conda executable for your installation of conda ("auto" will suffice in many cases) will be the extent of necessary customization. If no conda executable is available on the system, R will install miniconda and establish an appropriate environment. See the Technical Details > Python section of the DIMSpec User Guide for more information.

In summary, a "reticulated rdkit" environment includes at minimum python 3.8, reticulate 1.24, and rdkit 2021.09.4; these are provided in an `/inst/rdkit/environment.yml` file for convenience of those who want to establish an environment independently. Once the environment is established, systems should be able to identify it without difficulty during the environment establishment phase. The easiest method to activate this integration (outside of sourcing the compliance file with `USE_RDKIT = TRUE`) is to call `rdkit_active(make_if_not = TRUE)` which will attempt to bind to the environment whose name matches `PYENV_NAME` and, if one does not exist, will create it. More support for python integration is not provided here as systems can vary considerably.

Plumber Application Programming Interface (API)

The `plumber` API service is available from within the project or by launching one of the attached shiny applications from the command line. A few settings of note may be of interest; these will always be pulled into your session when the compliance file is sourced. Within the `/config/env_glob.txt` file, the following will determine availability as well as how the plumber API is launched:

1. `USE_API` must be a logical scalar, by default set to `TRUE` to activate the API service in a background process;
2. `API_LOCALHOST` must be a logical scalar, by default set to `TRUE` to launch the API on your local machine. Set this to `FALSE` and provide a value for `API_HOST` to make the API service available on your network (if `API_HOST` remains blank, your machine's network identifier as pulled by `Sys.info()[["nodename"]]` will be used preferentially, which may or may not resolve on your network);
3. `API_HOST` must be a character scalar representing the resolveable network address (either IP or domain-based) of the host machine; this is ignored if `API_LOCALHOST` is set to `TRUE`;
4. `API_PORT` should be an integer specifying an open communication port on the host machine, by default set to `8080`; and
5. `INFORMATICS` must be a logical scalar, by default `TRUE` determining whether or not to activate chemometrics integration on the plumber service; set `USE_RDKIT` to `TRUE` (the default) to enable the endpoints that ship with the project (note: both are required for full functionality of the MSMatch application in its current form).

If `USE_API` is set to `TRUE`, the following more specific settings that may be of use for the `plumber` service are located in the `/inst/plumber/env_plumb.R` file:

1. `PLUMBER_OBJ_NAME` must be a character scalar determines the name of the service watcher in the process from which the API service is launched. By default this is `plumber_service` and will be a `callr::r_bg` object that can be used to interrogate the service (e.g. `plumber_service$is_alive()`);
2. `r_scripts` must be a character vector and determines which R files are sourced to make their functions available to the API definition file; and
3. `LOGGING_ON` must be a logical value determining whether to activate logging for the API; additional logging settings immediately follow this setting in the file.

In most cases, the `api_endpoint` function should be the entry point from R for getting data from the plumber API defined at `PLUMBER_URL` in your session environment.

It may also be desirable to launch a second service (e.g. for testing new endpoints or changes to existing ones, or for providing a primitive form of load balancing). To do this, set the `pr` parameter of `api_reload` to a different name (character scalar) and the `on_port` parameter to an open port (it will fail if the port is already in use). If the compliance script was run with `USE_API = FALSE` and `api_reload` is not available, it may be more intuitive instead to use `start_api`, which will establish the service as if the compliance script had been run. The `api_endpoint` function defaults point to the URL established for the primary API service host; if more than one plumber service is running, set parameter `server_addr` more specifically to reference the desired endpoint complete with port (e.g. `api_endpoint("_ping", server_addr = "http://127.0.0.1:8181")`).

See more details in the Plumber section of Technical Details in the DIMSpec User Guide or the DIMSpec Quick Guide - Plumber vignette.

Shiny Web Applications

Perhaps the most common use case for using parts of DIMSpec will be developing additional user interfaces with custom functionality on top of the DIMSpec infrastructure. Several helper functions for shiny development in DIMSpec are also available in the `/inst/apps/shiny_helpers.R` file to abstract some common shiny UX tasks. When a DIMSpec R session is established through the compliance script, if `USE_SHINY` is set to `TRUE` a named vector is available as session object `SHINY_APPS` enabling you to quickly launch apps quickly using `start_app("X")` where `"X"` is a name in this object. Most settings and functions supporting shiny apps assume users will be familiar with developing in shiny.

To launch shiny apps on your local network and make those apps easily available to others, the environment resolution pathways for DIMSpec allow direct launching from any R terminal using `shiny::runApp("X", host = "0.0.0.0", port = Y)` where `"X"` is the path to a shiny application and `Y` is an integer representing an open port on your host machine; this should be executed from the project directory. (Note this is not an advanced feature limited to DIMSpec.) If `port` is not provided, a random open port will be assigned; while this is convenient for testing, or consistency it is recommended to set the port for each application. Other hosting options are available for shiny apps; your organization may have preferences which should be honored.

Settings for shiny applications are generally best left to the individual applications to resolve as needs may vary considerably. Given the integration of plumber with the DIMSpec project however, an environment resolution script is provided at `/inst/apps/env_shiny.R` which does have the following settings of note:

1. `packs` determines which packages will be available to all apps launched from DIMSpec and will install them if necessary;
2. `LOGGING_ON` is set to `TRUE` here by default to assist with debugging shiny applications;
3. options `plumber.host` and `plumber.port` are set from the `API_HOST` and `API_PORT` settings in `env_glob.txt` settings (described above) to link launched apps with their defined plumber APIs;
    - If your application requires a separate API it is recommended instead that in the `global.R` file of your app, you set `USE_API = FALSE` and these options there along with `PLUMBER_URL` as the server address desired; this will override the environment settings and point your new application to the endpoint you define. This does require you establish the API availability ahead of time.
4. `logger_settings` will establish the logging parameters for the launched application and by default will copy settings from the `LOGGING$SHINY` object if available.

To build new shiny web applications in an active DIMSpec project, a templated directory has been provided at `inst/apps/app_template/` and contains a skeleton similar to other web applications included in DIMSpec. In our experience, this app template greatly shortens many technical aspects of development as it leverages established environments and connections to the plumber API. The MSQC application was built from this template in a fraction of the time it would have taken when starting from scratch.

All other settings are left to the individual applications to define needs. See more details in the Shiny Applications section of Technical Details in the DIMSpec User Guide or the DIMSpec Quick Guide - Shiny Web Applications vignette for details of using the included application template.

Final Words

Due to its nature, the DIMSpec project includes several aspects, some complicated. The quick guides are meant as a, hopefully, brief reference for some of the more common use cases and settings, and should be viewed as supplements for the full DIMSpec User Guide. A note on network hosting: if your organization (as it should) scans for active sites and vulnerabilities, and you are running the plumber API or shiny applications on your local network in the manner described here, contact your IT professionals to ensure adherence to your organization's IT security policies.

The DIMSpec team hopes the documentation provided for the project is sufficient for your needs and allows you to quickly iterate. We want the project to be successful; reach out by email at pfas@nist.gov with ideas or contributions.

This concludes the Quick Guide for Developing with DIMSpec.

References

Adusumilli, Ravali and Mallick, Parag. 2017. "Data Conversion with ProteoWizard msConvert." *Methods in Molecular Biology (Clifton, N.J.)* 1550: 339–68. https://doi.org/10.1007/978-1-4939-6747-6_23.