Extending trackr with custom backends

Gabriel Becker

February 19, 2018

Contents

1 Creating custom backends **2**

2 Customizing metadata extraction **5**
2.1 Adding custom tags for a class 5
2.2 Defining customized formal metadata for a class 6

1 Creating custom backends

The `trackr` package supports storing metadata and artifacts in either a JSON file or Solr database. To extend `trackr` to support a new storage backend, it is necessary to define a derivative of VirtHistoryTracker and implement methods on six generic functions:

1. prep_for_backend, which prepares a PlotFeatureSet for insertion into the backend data store;
2. insert_record , which inserts the prepared feature set into the database;
3. remove_record, which removes a plot from the database;
4. trackr_write , which writes the the current state of an in-memory backend to disk;[1]
5. trackr_lookup , which looks up a single plot based on its ID or an object representing it; and
6. trackr_search , which provides whatever search functionality is supported by the backend.

These functions are called internally by the user-facing addPlot, rmPlot, and findPlot functions.

For the remainder of this section, we will walk through the creation of a toy-example backend based on **R** lists. While this would not suffice in practice because it offers no persistence, it will serve as an illustration of how one would implement a backend more generally — e.g., for MongoDB [**?**] or your database of choice.

To implement our backend, we first define the S4 class representing an instance of our list backend, ListBackend, which simply inherits from the base list class. We also define a constructor, for convenience.

```
library(trackr)

lb = setRefClass("BasicListBackend", fields = list(dat = "list"))
BasicListBackend = function() lb$new()
```

We then define our methods which will allow `trackr` to interact with our backend. For full customization, we could in principle first write a specific prep_for_backend method. When writing custom prep_for_backend methods, they must:

- Generate a unique id for the object, typically by calling uniqueID on the ObjectFeatureSet ;
- use trackr_lookup to confirm no plot with that id already exists in the datastore;
- write any image-files and serialized objects that need to be created in the file system;
- transform the plot, metadata, and file paths into the form your insert_record method will be able to process; and
- return the transformed data

In practice, however, `trackr` provides an "ANY" method which will generally suffice for most backends. It returns a list with a generated id and a flattened list of the plot's metadata, along with saving out image files based on the options set on your TrackrDB. We will use this method by simply not creating a backend-specific one for our ListBackend class. With prep_for_backend addressed, we move on to methods for inserting plots into and removing them from our list data store.

All insert _record and remove_record methods must accept five and four arguments, respectively:

1. object - the document to be inserted or deleted within the datastore, or the id of the record to be deleted;
2. id - the unique id to associate with the added plot, or the id of the plot to remove *insert_record only*);

[1] This function can often be a no-op if insert_record performs a permanent write operation

3. target - the backend object (of the new class that we have created);
4. opts - a TrackrOptions object
5. verbose - a logical indicating whether the backend should emit informational messages (if supported)

The methods themselves must then update the backend object by performing the desired insertion or removal, assign the updated backend back into the database, and return the database. In our toy list-backend case, we can achieve this like so:

```
setMethod("insert_record", c( target = "BasicListBackend"),
function(object, id, target, opts, verbose = FALSE) {
    target$dat[[id]] = object
    invisible(target)
})

## [1] "insert_record"

setMethod("remove_record", c(object = "character", target = "BasicListBackend"),
function(object, target, opts, verbose = FALSE) {
    ## object is the id
    target$dat[[object]] = NULL
    invisible(target)
})

## [1] "remove_record"
```

Note we are only defining a remove_record method for when the object argument is a character, thus our backend only supports removal by id. The `trackr` package uses S4 dispatch to funnel all logic through character-based id methods for remove_record, so no other methods are required.

Along with our insert_record and remove_record methods, we must define a trackr_write method which finalizes any changes to the `trackr` database our backend uses. The framework requires this in order to allow backends to accumulate changes in memory or immediately write them (within insert_record and remove_record, in which case trackr_write is a simple no-op).

Because our list backend has no persistent storage behind it, our trackr_write method does nothing:

```
setMethod("trackr_write", c(target = "BasicListBackend"),
function(target, opts, verbose = FALSE) target)

## [1] "trackr_write"
```

Next is a trackr_lookup method, which takes the object, db, backend and exist parameters. The object, put simply, is the query. The db and backend parameters represent and behave as they have throughout this discussion. Finally, the exist parameter specifies whether the lookedup plot entry from the data-store (FALSE — the default) or whether a logical value indicating whether such an entry was found (TRUE) should be returned.

The object argument can be an R object representing, an ObjectFeatureSet object, or a character id. As with remote_record, however, general methods provided by `trackr` funnel all dispatch through character-based methods, so we only need to provide a character id-based lookup method when defining a backend. We implement this like so:

```
setMethod("trackr_lookup", c("character", target = "BasicListBackend"),
function(object, target, opts, exist = FALSE){
    found = which(object == names(target$dat))
    if(exist)
```

```
        return(length(found) > 0)
    else
        return(backend[[found]])
})

## [1] "trackr_lookup"
```

Finally, the last method we require to define our backend is trackr_search . This accepts a regular expression (pattern), the usual db and backend, a vector of fields to search (fields), the form in which to return the results (ret_type — id, list, or a backend-specific option), and a verbose option. Its exact behavior may be backend-specific, but it is intended to find matches to pattern within the selected fields within the backend's datastore.

In the case of our backend, trackr_search will loop through our list and grep for the supplied pattern. We will write a simplied version which always returns the vector of matching ids:

```
setMethod("trackr_search", c(pattern = "character", target = "BasicListBackend"),
          function(pattern, target, opts, fields = NULL, ret_type = c("id", "list", "backend
                   verbose = TRUE) {
    if(is.null(fields)) {
        fields = TRUE ## grab all of them
    }

    inds = sapply(target$dat, function(y)  any(grepl(pattern, paste(y[fields]))))
    names(target$dat)[inds]
    })

## [1] "trackr_search"
```

With this, our backend is complete and ready for use.

```
library(ggplot2)
plt = qplot(x = 1:10, y = rnorm(10))
be = BasicListBackend()
db = TrackrDB(backend = be)
defaultTDB(db)

## An object of class "TrackrDB"
## Slot "opts":
## An object of class "TrackrOptions"
## Slot "insert_delay":
## [1] 0
##
## Slot "img_dir":
## [1] "./images"
##
## Slot "img_ext":
## [1] "png"
##
## Slot "backend_opts":
## list()
##
##
```

```
## Slot "backend":
## Reference class object of class "BasicListBackend"
## Field "dat":
## list()

record(plt)

## Warning in fullData(object): Only plotted data will be returned, which may be
limited to a summary of the original data. This may be due to use of vectors (rather
than a data.frame) in the ggplot call.
## Creating image directory at ./images

findRecords("rnorm")

## [1] "SpkyV2_c8927e6419bddbd7b4ba3e347f63eb62"
```

In this section, we provided methods for all low-level generics (other than prep_for_backend, which is only required in special cases), for the purposes of illustrating the extension mechanism. In point of fact, however, `trackr` provides default methods for insert_record , remove_record, and trackr_lookup which work for any backend whose class has [[and [[<− methods defined which allow indexing by character, and where assignment of NULL into an existing entry is equivalent to removal. Thus, only trackr_write and trackr_search methods are actually required in many cases, including the list -backend case we presented here.

2 Customizing metadata extraction

2.1 Adding custom tags for a class

Metadata extraction can be customized in two ways within the `trackr` framework. First, users can define an S4 method for the generateTags generic for a class. This is called by makeFeatureSet methods `trackr` provides to perform metadata extraction, and is passed the object that `trackr` is extracting metadata from. Any (character) values returned will be added to the tags for the object. This provides an easy way to add non-key-value annotations to objects you are recording.

NOTE: defining an S3 method (simply creating a function called generateTags . yourclass will not work. You must create a formal method.

```
y = 5
class(y) = "sillyclass"
setMethod(generateTags, "sillyclass", function(object) "Hi vignette readers!")

## in method for 'generateTags' with signature '"sillyclass"': no definition for
class "sillyclass"

## [1] "generateTags"
## attr(,"package")
## [1] "trackr"

fs = makeFeatureSet(y)
tags(fs)

## [1] "Hi vignette readers!"
```

2.2 Defining customized formal metadata for a class

Customizing tag generation only allows us to add tags, however. We can also customize the formal key-value metadata pairs generated for a class of object. This is a two-step process. First we extend the virtual FeatureSet class with a subclass that adds slots for the formal pieces of metadata we will extract from objects of a particular class. Often we will actually inherit from ObjFeatureSet, which has slots for the object itself and its class. We will create a toy example which customizes the metadata extracted from integer vectors by capturing the set of unique values the vector takes.

```
setClass("AwesomeIntFeatureSet", contains = "ObjFeatureSet",
         slots = c(uniquevals = "integer"))
```

We then define a method for makeFeatureSet that takes an integer vector and returns an AwesomeIntFeatureSet object. How the internals of the method behave is up to the author, but suggested practice when inheriting from a subclass of FeatureSet — as we are here by inheriting from ObjFeatureSet — is to use the constructor for the parent class to generate any non-customized information, then use that in construction of the final object, like so.

```
setMethod("makeFeatureSet", "integer", function(object, ...) {
    innerobj = ObjFeatureSet(object, ...)
    new("AwesomeIntFeatureSet", innerobj, uniquevals = unique(object))
    })

## [1] "makeFeatureSet"

x = sample(1:20, 20, replace=TRUE)
makeFeatureSet(x)@uniquevals

##  [1] 10 12  2  1  7  3  4  6 13 18
```