What is beakr?

beakr is an unopinionated and minimalist web framework for developing and deploying web services with R. It is designed to make it as simple as possible for data scientists and engineerings to quickly write web applications, services, and APIs without worrying about lower-level details *or* high-level side-effects. In other words, **beakr** is made to be *explicit, robust*, and *scalable* – and the batteries are not included.

beakr is built on the **httpuv** package – which itself is built on top of the libuv and http-parser C libraries. So while **beakr** will be stable for most use cases, it is not necessarily recommended for building extensive web-applications and is not designed to be an especially performant web framework. If you're looking for a full-featured web framework see Shiny, django, *etc*. **beakr** is inspired by the minimalist and massively-expandable frameworks offered by Express.js and Flask.

beakr takes a programmatic approach to creating web-services and the framework focuses on just core HTTP protocol functionality (*e.g* POST, GET, PUT, *etc.*). To illustrate, we can create a "Hello World" application with **beakr** with only a few lines of code.

```
library(beakr)
beakr <- newBeakr()
beakr %>%
  httpGET("/hello", function(req, res, err) {
    "Hello, world!"
  }) %>%
  listen()
```

Running this script will serve the application (by default on `host = 127.0.0.1` and `port = 25118`). When a client points to the path `/hello` a GET request is made to the server. The **beakr** instance is *listening* and handles it with a defined server response – in this case, the response will show an HTML page with the plain-text "Hello, world!".

So what is beakr good for?

beakr is simple and flexible and as such is best suited for simplicity. It is a great tool to quickly and easily stand up web services without the limitations of heavier-weight contenders so you can worry less about having to learn a new framework and more about what the application should do. A few examples could be:

- Building RESTful APIs
- Building web-applications and services

Examples

REST API

It is increasingly important for data scientists to share operational data across computers, languages, and pipelines. **beakr** makes it easy to create micro-services like RESTful (REpresentational State Transfer) APIs and other infrastructure that are flexible and easy to deploy, scale, and maintain.

For example, we can easily build a RESTful API with two endpoints (`/Sepal.Length,`

`/Sepal.Width`) that will respond with a JSON data object retrieved from from Fischers Iris dataset.

```
library(beakr)

data("iris")

beakr <- newBeakr()

beakr %>%
  httpGET("/iris/Sepal.Width", function(req, res, err) {
    res$json(iris$Sepal.Width)
  }) %>%
  httpGET("/iris/Sepal.Length", function(req, res, err) {
    res$json(iris$Sepal.Length)
  }) %>%
  listen()
```

Now a client (or a web-browser) can access the JSON data at 127.0.0.1:25118/Sepal.Length and 127.0.0.1:25118/Sepal.Width.

Deploy a model

beakr can also be used to deploy models and algorithms as micro-services. For example, we can deploy a simple KNN-model to return the flower species (from the Iris dataset) to a client as plain text. Using the POST method we can create a **beakr** instance to handle recvieving a JSON object containing the sepal and petal lengths and widths (`sl, sw, pl, pw`, respectively).

First we can define and train a simplistic K-nearest neighbors flower model using the **caret** package.

```
library(caret)

# Load the Iris data set
data('iris')

# Train using KNN
knn_model <- train(
  Species ~ .,
  data = iris,
  method = 'knn',
  trControl = trainControl(method='cv', number=10),
  metric = 'Accuracy'
)
```

We can create and expose normal R functions using **beakr**'s built in `decorate()` function, which easily prepares functions to accept parameters and respond as you'd expect. With this is mind, we can write a simple function to accept our petal and sepal parameters and return the predicted species using our model defined above.

```
# Function to predict the species using the trained model.
predict_species <- function(sl, sw, pl, pw) {
  test <- data.frame(
```

```
    Sepal.Length = as.numeric(sl),
    Sepal.Width = as.numeric(sw),
    Petal.Length = as.numeric(pl),
    Petal.Width = as.numeric(pw),
    Species = NA
  )
  predict(knn_model, test)
}
library(beakr)

# Use beakr to expose the model in the "/predict-species" url path.
#   See help("decorate") for more info about decorating functions.
newBeakr() %>%
  httpPOST("/predict-species", decorate(predict_species)) %>%
  handleErrors() %>%
  listen(host = "127.0.0.1", port = 25118)
```

By sending a POST request to 127.0.0.1:25118/predict-species with a JSON object containing labeled numeric data we can see it responds with an answer.

```
$ curl -X POST http://127.0.0.1:25118/predict-species
  -H 'content-type: application/json'
  -d '{ "sl": 5.3, "sw": 4, "pl": 1.6, "pw": 0.2 }'

> setosa
```

beakr really shines in these sort of things. At Mazama Science, we use **beakr** to create a large variety of web services that perform tasks like:

- creation of raw and processed data products
- creation of data graphics for inclusion in other interfaces
- automated report generation

Any task that you might create a script for is a candidate for conversion into a web service. **beakr** has become a proven tool that we at Mazama Science are excited to share!

You can install **beakr** from CRAN with: `install.packages('beakr')`.