`rserve-ts`: a `TypeScript` interface to `Rserve` for modern web apps

Elliott, T.M.

iNZight Analytics Ltd

University of Auckland

Abstract

1 Introduction

The `Rserve` package (Urbanek, 2003) provides a way for programming environments to call R functions.

2 Typescript

Typescript is a language for writing type-safe Javascript applications. It allows developers to define types for their packages which makes both development for them and others easier. It also ensures that almost all possible runtime errors are caught while writing the code. Additionally, development features like autocomplete make developing, extending, and debugging Typescript applications significantly easier. As a result, most modern web applications are now built using Typescript on top of whatever framework is being used (e.g., React, Svelte, etc).

The `rserve-js` package has received very little development, despite being stable and actively used in the RCloud project. This means that it still employs ancient[1] Javascript coding style, including instantaneously invoked function expressions (IIFEs) and no type-safety. It also uses callback expressions for asynchronous

[1] 10-year old

code, instead of promises which are more useful from an application programming interface (API) standpoint. To bring `Rserve` to modern web development, we created a new package `rserve-ts` which provides a modern API to `Rserve`.

The key points to discuss are:

- callback vs promises
- known return types
- user-specified value types (which must be asserted at runtime)

2.1 Asynchronous functions: callbacks and promises

The `rserve-js` library is written using callbacks, an increasingly uncommon design choice among more modern libraries. Callbacks allow code to be executed once a function has completed evaluation, and is critical for between-environment communication; otherwise the client would appear frozen to a user while the evaluation is performed. As an example, let's say we have a function that downloads a file from a URL and copies the data into a database.

```
const loadData = (url: string) => {
    // download and save data
    return true;
}
```

This may take some time to complete, so in modern Javascript we would generally write this as an *asynchronous* function by using the `async` keyword:

```
const loadData = async (url: string) => {
    // download and save data
    return true;
}
```

When this function is called somewhere in our code, Javascript knows that it cannot expect the result immediately, and will continue executing the following lines of code before `loadData()` is finished. Instead, the `async` keyword lets the function return a *promise* instead, of the form "the result will eventually be a boolean, but I haven't finished yet". Once the promise is completed and the value is finally returned, any

following code can be executed. There are two ways of doing this with promises: `then()` or `await`. The former provides code that will be run appended to the promise, like so:

```
loadData('http://...').then(() => console.log('Data loaded!'));
// more code that will run before loadData() completes
```

This is good if the 'more code' does not depend on `loadData`, for example it may clear away any forms in the graphical user interface (GUI) or provide some loading information to the user.

Sometimes, however, it might be preferred if the function waits until the asynchronous one is complete. This is useful when there is a chain of asynchronous functions one after another, and multiply-nested `then()`'s can become complex. This alternate uses the `await` keyword, and additionally requires that the wrapping function is itself an async function, like this:

```
const main = async () => {
    const res = await loadData('http://...');
    if (!res) console.log('Data failed to load');
    console.log('Data loaded!');
}
```

Here, we wait until `loadData()` has finished executing and the promise is full-filled. Then we can check if the result is indeed `true`, and handle it accordingly.

This is a very nice and tidy way of operating with asynchronous functions in Javascript, as the async/await syntax is particularly useful for widgets that depend on data or other long-running computations. However, `rserve-js` is written using callbacks, which are more like then `then()` syntax but instead of appending this after the promise, it is passed as an argument to the function itself. That is, the function expects a callback function that it will execute once it has finished.

```
 const loadData = (url: string, cb: (err: string | null, res: boolean) => void) => {
    // download and save data
    cb(true);
}


loadData('http://...', (err, res) => {
    if (!res) console.log('Data failed to load');
    console.log('Data loaded!');
```

```
})
```

2.1.1 Handling known return types

2.2 User-specified types from R

This is the trickiest part of this project. Requires developing a syntax that maps between R's front- and back-end data structures, `rserve-js` data structures (which mimic the back-end structures) and a friendly API for developers that mimic's R's front-end API.

In Typescript, *generic functions* allow users to either pass in objects of a known type (which gets inferred by the function), or specify the type of the thing they are passing out or, in our case, expecting back. The syntax for this is

```
const x = myfun<{ result: number[] }>()
```

For example, the R object `c(1, 2, 3)` is a numeric vector. This is represented as a complex `SEXP` (which is the term used to describe an arbitrary R object, based on the predecessor language S). If you look at the result in Javascript, you get this:

```
{
  type: 'sexp',
  value: {
    type: 'double_array',
    value: Float64Array(3) [ 1, 2, 3 ],
    attributes: undefined
  }
}
```

Requiring users to define even this simple object would be overkill. So instead, we came up with a set of *type helper functions* which can be used to express R objects as Typescript types, and infer the underlying `rserve-js` structure.

```
import { Numeric, Character, Boolean, Dataframe, List } from 'rserve-ts';

type myVector = Numeric;
type myDataFrame = Dataframe<{
    x: Numeric,
    y: Character,
    z: Boolean
}>;
```

```
type myListWithAttributes = List<{
    x: Numeric
}, {
    label: string,
    checked: boolean
}>
```

Due to the complex way in which R objects can be built, we had to make use of Typescript's inference and generic methods to come up with an API that is simple yet flexible enough to handle everything. Table X shows the base types available in `rserve-ts` and their mappings back to `rserve-js`.

The difficult aspect is distinguishing between constant and unknown values. For example, a factor has (potentially) known levels, but unknown values. A character vector may or may not be known: if it represents the names of something, or options for a function argument, for example, then the values are known and therefore this should be reflected in the type definition.

As an example, compare the types of these two values:

```
const x = ['one', 'two', 'three'];
// x: string[]

const y = ['option1', 'option2'] as const;
// x: readonly ['option1', 'option2'];
```

The `readonly` means that this string array is fixed and cannot be modified.

Writing a type helper for a normal character vector is fairly straightforward. Here it is, in fact:

```
type Character = {
  type: "character_array",
  value: string[],
  attributes: undefined
}
```

If we want to allow users to optionally pass in the known values, we can make the type helper *generic*.

```
type KnownCharacter<T extends string[]> = {
  type: "character_array",
  value: T,
  attributes: undefined
}
let levels: KnownCharacter<["one", "two"]>;
// levels.value: ["one", "two"]
```

Having to use different helpers in these similar situations adds unnecessary complexity. We can simply combine the two by adding in a default value.

```
type Character<T extends string[] = string[]> = {
  type: "character_array",
  value: T,
  attributes: undefined
}
let vec = Character;
// vec.value: string[]
let names = Character<["one", "two", "three"]>;
// names.value: ["one", "two", "three"]
```

The last piece of the puzzle is the presence of the `json()` method on the `rserve-js` types. This allows users to obtain a simple JSON type of the object, rather than the more complicated structure shown so far.

```
const x: Character = {
    type: "string_array",
    value: ["one", "two", "three"],
    attributes: undefined
}
x.json()
// ["one", "two", "three"]
```

In an app, the last form is obviously our preferred way of working with the data.

A slightly annoying thing about the `json()` method is that in many cases it returns a scalar if the length of the array is 1.

3 Types

3.1 Base types

The base types include boolean, integer, double (numeric), and character types. The `rserve-js` implementation of these types introduces a small complexity whereby *scalar* values are returned as scalars, rather than as arrays of length 1 (which would drastically simplify everything). Therefore, it is necessary to tell Typescript the expect length (1 is a scalar, any other value is an array, including 0).

Scalar values are exactly the single value with no additional attributes. Array values are arrays or typed arrays (e.g., `Float64Array`) with the additional properties `r_type` and `r_attributes`.

To allow these values, we had to rely on the `z.custom()` method from the `zod` package, as there is presently no way to represent an array with additional properties.

3.1.1 Logical/boolean

These are vectors of `TRUE` and `FALSE` values in R. They are represented as an array of booleans in Javascript.

```
console.log(await R.eval('c(TRUE, FALSE, TRUE)', R.logical(3)));
// [ true, false, true, r_type: 'bool_array' ]
console.log(await R.eval('1 == 2', R.logical(1)));
// false
```

If the length is unknown, the user will have to check the result:

```
const whichery = await R.eval('which(x == 3) < 5', R.logical());
if (typeof whichery === "boolean") {
    console.log('There', whichery ? 'is' : 'is not',
      'a value of 3 in the first 5 elements of x');
} else {
    console.log('There are', whichery.length,
      'values of 3 in the first 5 elements of x');
}
```

This all changes, however, if there are attributes attached to the vector, in which case the array result is always returned:

```
console.log(await R.eval('structure(TRUE, label = 'myvector')',
  R.logical({ label: 'myvector' })));
// [ true, r_type: 'bool_array', r_attributes: { label: 'myvector' } ]
```

In an future R package we will provide a result type that adds attributes to results, which will allow users to easily ensure a result is always an array.

3.1.2 Integers and Floats

These are vectors of integers and floats in R (using either `integer()` or `numeric()`, or the shorthand `1L`, etc). They are represented as Typed Arrays in Javascript.

```
console.log(await R.eval('1:3', R.integer(3)));
// Int32Array(3) [ 1, 2, 3, r_type: 'int_array' ]
console.log(await R.eval('c(1, 2, 3)', R.numeric(3)));
// Float64Array(3) [ 1, 2, 3, r_type: 'double_array' ]
```

Again, users should be careful when the length is unknown, as these are returned as scalars.

3.1.3 Strings, characters

Vectors of strings in R are represented as arrays of strings in Javascript.

```
console.log(await R.eval('hello world', R.character(1)));
// 'hello world'
console.log(await R.eval('c("a", "b", "c")', R.character(3)));
// [ 'a', 'b', 'c', r_type: 'string_array' ]
```

3.2 Vector types

Not to be confused with R vectors (which is effectively all objects), behind the scenes vectors are simply a collection of values. The most primitive example in R is `list()`, but can include tagged lists (which are lists with names), data frames, and attributes.

3.2.1 Lists

Lists in R are represented as *vectors* (`XT_VECTOR`) which can have names (tags). These are represented as either arrays or objects in Javascript, depending on whether the list has names or not.

```
console.log(await R.eval('list(a = 1, b = 2, c = 3)', R.list()));
// { a: 1, b: 2, c: 3, r_type: 'tagged_list' }
console.log(await R.eval('list(1, 2, 'hello')', R.list()));
// [ 1, 2, 'hello', r_type: 'vector' ]
```

Of course, the example above is just a simple demonstration of the concept. The values within a list can be any other valid R object. We can describe to Typescript the structure of the list in one of two ways:

```
1 type MyList = R.list<{ a: R.boolean(1), b: R.character(0), c: R.numeric(2) }>;
2 console.log(await R.eval('list(a = TRUE, b = c('hello', 'world'), c = rnorm(2))', MyList));
3 // {
4 // a: true,
5 // b: [ 'hello', 'world', r_type: 'string_array' ],
6 // c: Float64Array [ 0.728351, 0.262258, r_type: 'double_array' ],
7 // r_type: 'tagged_list'
8 // }
```

3.2.2 Data frames

Data frames are a special type of list in R where each element is a vector of the same length.

3.3 Other types

3.3.1 Factors

Factors are an important type of vector in R that have a fixed set of levels. Essentially, however, they are just integers with additional attributes.

```
console.log(
  await R.eval(
    'factor(c('a', 'b', 'c', 'b', 'b'), levels = c('a', 'b', 'c'))',
    R.factor(['a', 'b', 'c'])
  )
);
// [ "a", "b", "c", "b", "b",
// levels: [ 'a', 'b', 'c', r_type: 'string_array' ],
// r_type: 'int_array',
// r_attributes: {
//   class: 'factor',
//   levels: [ 'a', 'b', 'c', r_type: 'string_array' ]
// } ]
```

3.4 Attributes

Attributes are a way of adding metadata to an object in R. Essentially all objects can have attributes, and as mentioned earlier they effect the way the object is returned to the user (i.e., always the array type).

In Javascript, they are represented as an object with the attribute names as keys and the values as values.

Since the length is no longer important, we only need to pass the attributes.

```
console.log(
  await R.eval(
    'structure(1:3, labels = c('a', 'b', 'c'))',
    R.integer({ labels: R.character(3) })
  )
);
// Int32Array(3) [ 1, 2, 3,
//   r_type: 'int_array',
```

```
//   r_attributes: { labels: [ 'a', 'b', 'c', r_type: 'string_array' ] }
// ]
```

4 Working with types

The Typescript library `zod` is used to validate the types of the objects returned from `rserve-js`. This means that the user can be confident that the object they are working with will be of the correct type at runtime, regardless of what happens in R.

The caveat is that if an invalid type is returned, the program will throw an error (but at least it will be caught immediately rather than throwing up some arbitrary error later on).

The main advantage of this is that user experience (ux) is improved, allowing front end web developers to easily build out data-driven applications without having to worry about the data types they are working with.

5 Conclusion

References

Urbanek, S. (2003). Rserve – A Fast Way to Provide R Functionality to Applications. *Friedrich Leisch & Achim Zeileis.*