ReactRacer

v0.1.0

Bringing Racer knowledge to the browser

Getting started with ReactRacer

Overview

Most applications nowadays have migrated from the desktop version to a more accessible, web browser version. The React implementation offers a lightweight, easily extendable interface between the browser and the Racer server. Therefore, it cuts out the need of a desktop client to access the functionalities, which also gives more accessibility.

Prerequisites

To start using the current version of the application, there are several prerequisites that have to be satisfied:

- having a Node.JS server installed on the personal computer; any version is acceptable; the current implementation was made on version 16.14.0;
  This can be downloaded from the official page: https://nodejs.org/en/
- having a packet manager installed on the personal computer: the current implementation was made by using **Yarn** (it has some advantages over **npm**)
  Yarn can be installed from the official page:
  https://yarnpkg.com/getting-started/install
- having the Racer server running on the personal computer.
  This can be downloaded either from the official page, or from the GitHub repository (either requires CommonLISP to compile and run the program)
  https://www.ifis.uni-luebeck.de/~moeller/racer/download.html
  https://github.com/ha-mo-we/Racer/
- enjoying working with ontologies & knowledge bases

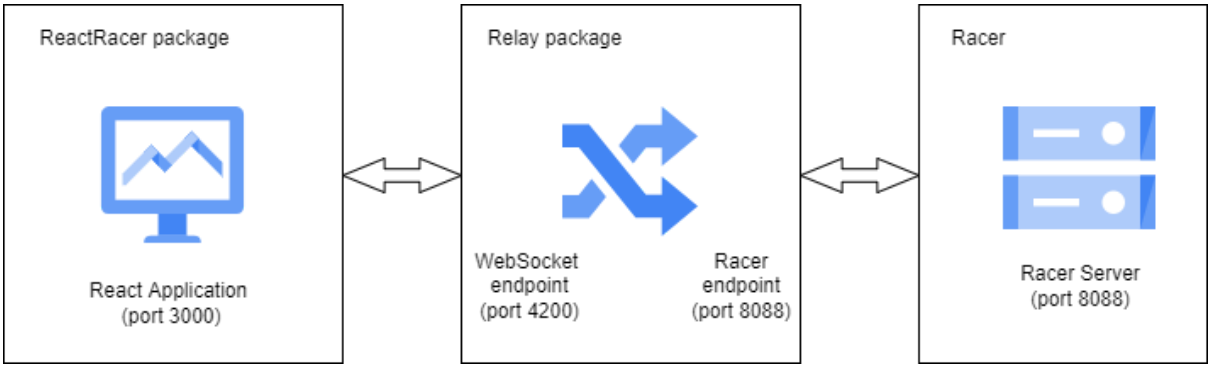
The two applications (ReactRacer and the relay server - described in the next section) are available open source on GitHub:

–insert github repo link –

Running the application

All the communication to the Racer server is done via TCP Sockets, as designed by its creators. Web browsers do not have the possibility to use this kind of sockets for security reasons. However, a particular and similar communication method is implemented for a browser - WebSockets. In order to connect the React application to the Racer server, all the requests have to be relayed through a third application that understands both TCP Sockets and WebSockets.

The project is structured as a monorepository: the relay in Node acting as the server and the React client.



First, open a terminal in the root folder of the project. Making use of Yarn workspaces, the management of the codebase was greatly simplified. To run the relay server, simply type in the terminal the following command:

yarn workspace relay start

```
PS C:\Users\radu.bouaru\Documents\ReactRacer> yarn workspace relay start
yarn workspace v1.22.17
yarn run v1.22.17
$ node WebsocketRelay.js
Relay server started on localhost, port 4200
```

Next, type in the command to open the React application:

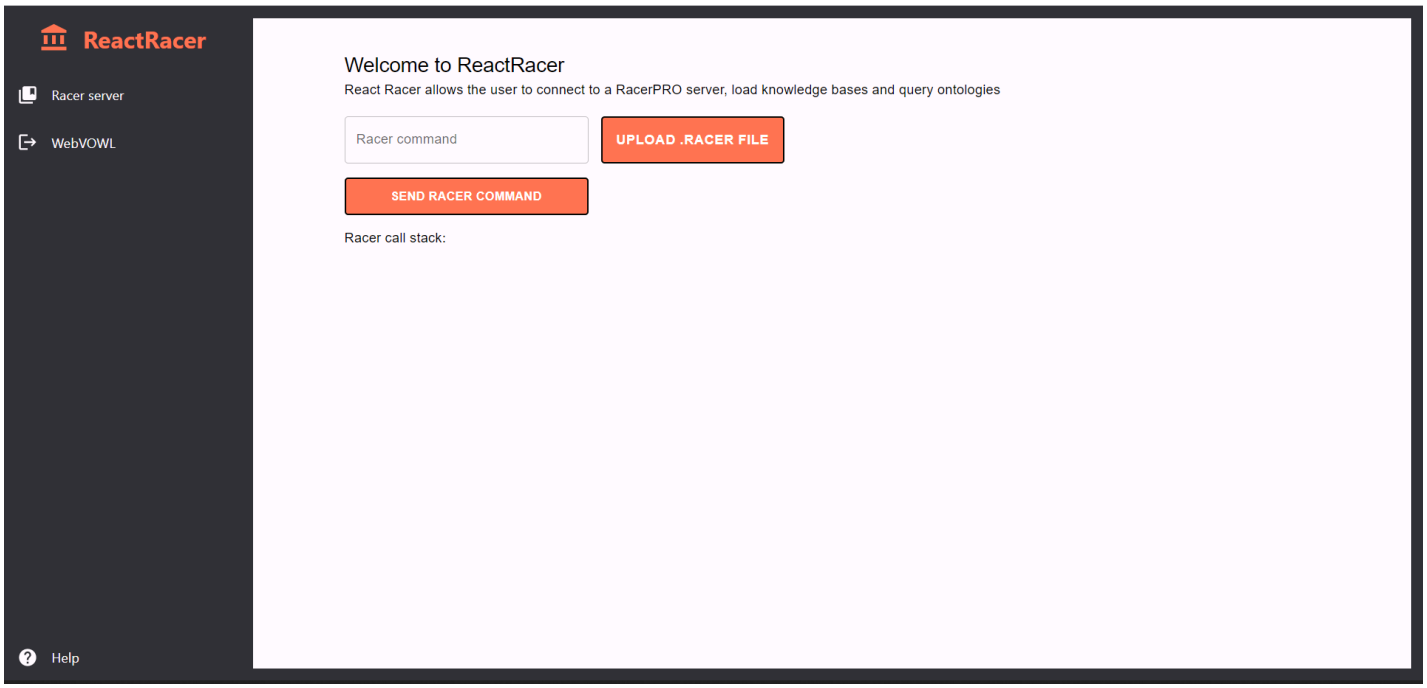
yarn workspace reactracer start

```
PS C:\Users\radu.bouaru\Documents\ReactRacer> yarn workspace reactracer start
```

```
Compiled successfully!

You can now view reactracer in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://172.31.64.1:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

Because this is a local environment, the production build is not required.

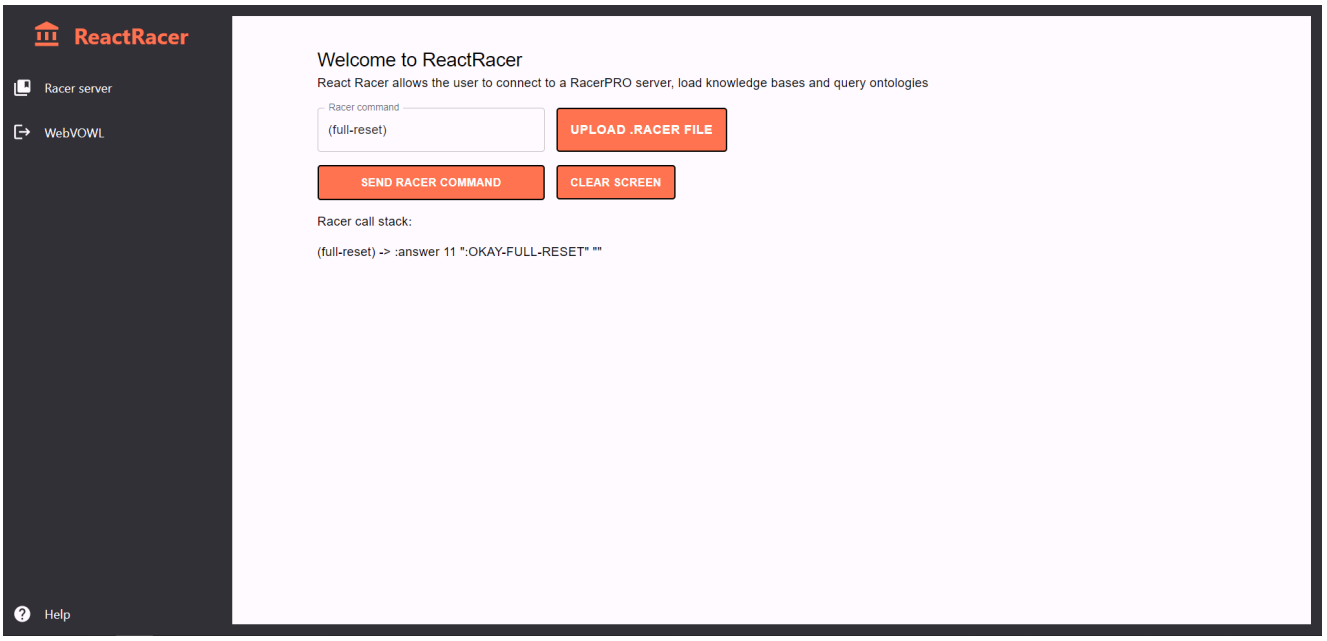
Features



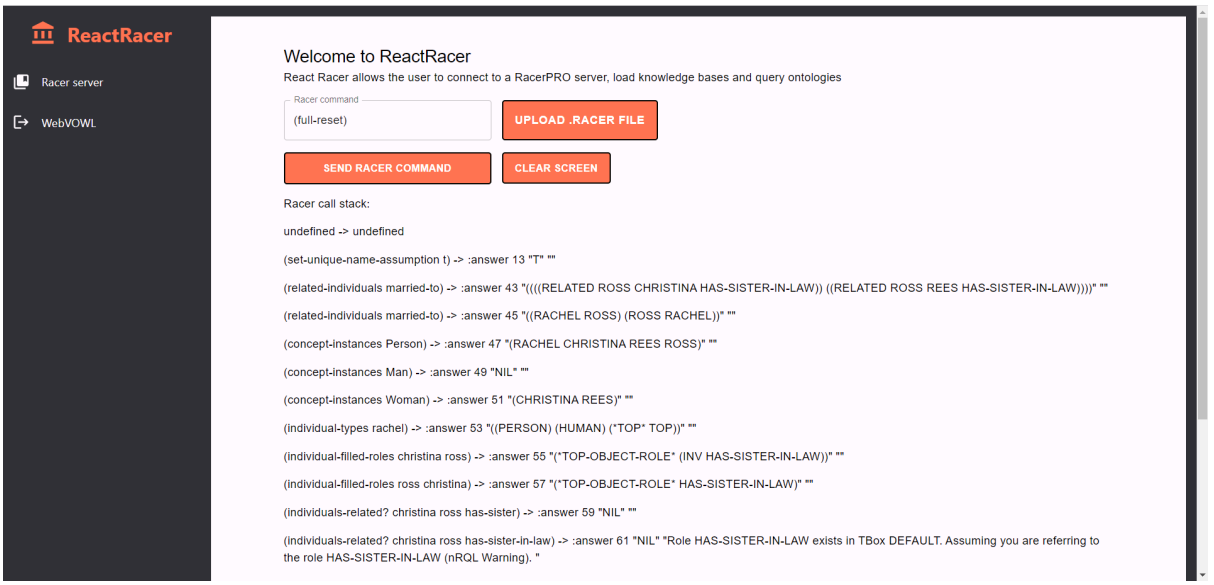
On the main screen, the basic functionality is presented.

The possibilities are:

- an user can enter RACER commands directly in the textfield and see the results directly below;



- an user and upload an ontology/knowledge base in the .racer format and see the results directly below;



- An user can click on the WebVOWL button in the left panel to be redirected to the WebVOWL website;
- An user can click on the Help button in the left panel to be redirected to the GitHub repository of the application;
- An user can click on the *Clear Screen* button when the page becomes too crowded with commands;

- When an user is scrolling down below, a button pops out on the screen, which allows the user to automatically scroll back to the top of the page.

Implementation details

First of all, the configuration of the hosts and ports is important. The Racer server TCP service runs everytime on localhost:8088 by default. There are two places where the user must modify the connection parameters, in order for the applications to work.

In WebsocketRelay.js, a user can configure the port of the relay server. This server is automatically run on **localhost**, too. The port value set here has then to be added in the React configuration file!

```
8   const server = http.createServer(app);
9   const port = 4200;
10  server.listen(port);
```

For the React application, in src/Configuration/configuration.js file, the following configuration settings are presented:

```
1  let configuration = {
2    relayUrl: "ws://localhost:4200",
3    remoteHost: "localhost",
4    remotePort: 8088,
5  };
```

The relay URL is the host for the relay. By default, it is initialized to be localhost:4200 (see the pictures above). The remoteHost and remotePort fields represent the host port of the Racer server. They are initalized as mentioned before.

By having these fields set up correctly, the next step is to run both the relay and the React applications.

Second, the React application checks if the relay is opened by submitting requests. This gives the user the possibility to open the server if he forgets to do so. The frontend offers warning and error messages if the connection between the client and the relay could not be initialized properly. The function that performs these checks can be found in src/SocketContext/SocketContext.js.

```
export const connection = async (socket, timeout = 100000) => {
    const isOpened = () => socket.readyState === WebSocket.OPEN;

    if (socket.readyState !== WebSocket.CONNECTING) return isOpened();
    else {
      const sleep = 100;
      const ttl = timeout / sleep;

      let loop = 0;
      while (socket.readyState === WebSocket.CONNECTING && loop < ttl) {
        await new Promise((resolve) => setTimeout(resolve, sleep));
        loop++;
      }
      return isOpened();
    }
  };
```

A socket context was created, so that the socket connection can be accessed globally through the React application.

The connections is set in the src/Components/ConnectionWrapper file. The flow follows this logic:

- a connection to the relay is initiated by the React client
- if the relay does not respond in a period of time, the connection is aborted and the client is prompted with a message
- if the relay responds, the connection is set
- the React client sends a message in the ***'open racer_host racer_port'*** format to the relay
- if the relay does not respond with any message, that means the Racer server is not opened and the user should first run the server
- if the relay responds with the ***'connected'*** message that means the relay connected to the Racer server
- all the links are properly initialized and the work can now begin

All the messages are sent via the socket as plain text - this could be a security issue if the ontology contains sensible information. The WebSockets implemented in JavaScript offer a method to send data and an event listener for the user to track any messages that might have been received from the server. The most basic method to send and receive information was implemented in the following manner:

```
const sendRacerCommandFromState = () => {
  let command = `${racerCall}\r\n`;
  if (racerCall !== "") socket.send(command);

  socket.onmessage = (event) => {
    setRacerMessages([
      ...racerMessages,
      { racerCall: racerCall, racerAnswer: event.data },
    ]);
  };
};
```

The Racer commands are sent using the socket.send() method. When the socket receives information from the server, the onmessage event listener fires the function defined after socket.onmessage. The received messages are kept in an array in which each entry has the following format: {racerCall, racerAnswer}.

As speed is not of a great concern in this application, some parts of the application were implemented in a synchronous manner - which is definetely not a good idea in general for a browser application. An asynchronous approach is preferred because it does not block the flow. This was required because the flow of data follows a complex path, and it must be somehow synchronized in order to obtain the correct results. Therefore, the operations that have been g ven temporal sense are reading commands line by line from a .racer file and sending those commands to the Racer server.

```
const readFile = async (file) => {
  let result = await new Promise((resolve) => {
    let reader = new FileReader();

    reader.onload = () => resolve(reader.result);

    reader.readAsText(file);
  });

  return result;
};

const onChangeFile = async (event) => {
  event.preventDefault();
  setRacerMessages([""]);
  const file = await readFile(event.target.files[0]);

  const lines = file
    .split("\r\n")
    .filter((line) => line.replace(/\s/g, "").length !== 0);

  setFileContent(lines);
};
```

```
useEffect(() => {
  async function load() {
    for (let i = 1; i < fileContent.length; i++) {
      sendRacerCommandFromParam(fileContent[i]);
      await timer(1);
    }
  }

  load();
}, [fileContent]);
```

After the file is read and converted to the required format (line by line), the effect is run and the commands are sent to the Racer server one each 1ms.

Any other piece of code not presented so far is simply React state management or basic UI building.

Limitations & known bugs

At this moment, the racer files that can be uploaded must have an instruction per line. Easy formats for human agents to read are not preffered by the parsing method. For example, a rule must be defined this way:

```
(define-rule (?x ?y has-sibling) (and (?x ?z has-grandparent) (?y ?z has-grandparent) (?z ParentOfOne)   (not (same-as ?x ?y))))
```

not this way:

```
(define-rule (?x ?y has-sibling)
        (and (?x ?z has-grandparent)
        (?y ?z has-grandparent)
        (?z ParentOfOne)
        (not (same-as ?x ?y))))
```

The same goes for any instruction. Another example:

```
(signature :atomic-concepts (Human Person Female Male Woman Man Brother Sister) :roles ((has-sister :domain Person :range Woman :transitive t :parent has-sibling)(has-brother :domain Person :range Man
:transitive t :parent has-sibling) (has-gender :feature t) (married-to :domain Person :range Person :symmetric t)(has-husband :domain Woman :range Man :parent married-to :inverse has-wife)(has-brother-in-law
:domain Person :range Man)(has-sister-in-law  :domain Person :range Woman)):individuals (rachel ross rees christina))
```

As long as they are one long line that contains the entire instruction, the program works as expected.

Another known bug is that the first sent instruction written in the textfield and sent always returns ILLEGAL SYNTAX answer. From the second answer on, everything works as expected. For files, this does not apply.

Tests

The application was tested among all the files presented by our teacher at the laboratory works & seminaries. Everything worked as expected - the results were compared to the results parsed in RacerPorter.

Ending notes

All the code is open-source, any improvements are highly desired. This code was also developed with the help of the open-source community. For more explanations, you can contact me on my email address: bouaru_radu@yahoo.com. Enjoy!