WIS-Client

Simple web page for REST calls, teaching project for purpose of practical works.

Authors

- Reto Schiegg
- David Wettstein

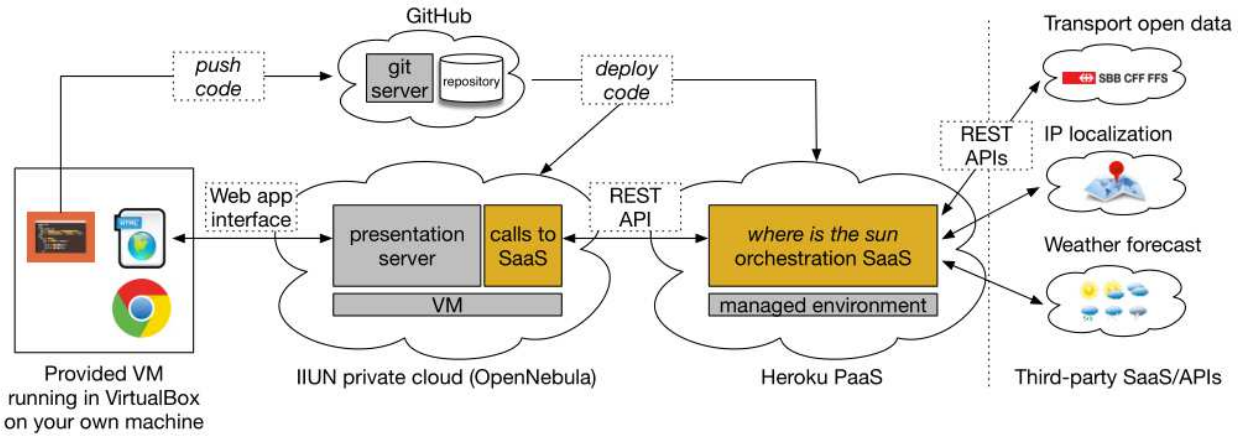
Git repositories

- WIS-Client: https://github.com/dwettstein/cloud-computing-2016-WIS-Client
- CCS-REST: https://github.com/dwettstein/cloud-computing-2016-CCS-REST

Production deployments

- WIS-Client: http://clusterinfo.unineuchatel.ch:10103/
- CCS-REST: https://leave-the-cloud-saas-app.herokuapp.com/

Description of the project



Setup

Used technologies

- Ruby (version 2.2.4)
- Sinatra
- Slim
- Capistrano

Run the CCS-REST app

Go into your repository location and either run the following commands in your shell:

```
bundle install
bundle exec puma -C config/puma.rb
```

or run the script `run.sh`.

Then go to: `http://localhost:4000`

Run the WIS-Client app

(Same as above...)

Then go to: `http://localhost:3000`

Deployment

For the deployment of the CCS-REST application, we are using Heroku and have set up an automatic build by connecting the GitHub repository. This is really handy and straight forward.

The WIS-Client application is deployed by using Capistrano, which is a remote server automation and deployment tool written in Ruby. For deploying run the following command (SSH keys are required for authentication):

```
bundle exec cap production deploy
```

RESTful-APIs

Our web service uses the following three APIs in order to get information about public transportation schedules, geographical data and weather conditions:

- IP-API.com - Geolocation API
- Transport Opendata CH - Swiss public transport API
- OpenWeatherMap

First, we implemented the in detail described IP request `/ip`. This route takes `ip` as a parameter and responses the location of the IP. As all further routes also have to forward the request to another API (Geolocation, Transport, Weather), we defined a function `forwardRequest` which makes a HTTP GET request to a given URI and returns its response as JSON.

After that, we implemented the route `/locations` which forwards all given parameters to the transport opendata API. However, either `query` or the coordinates `x` and `y` are mandatory otherwise an error message is shown. The logic of the routes `/connectoins` and `/stationboard` is implemented according to `/location` and does not need any further explanation. The next route is `/weather`. As web requests for current as well as future weather data are almost identical, we sourced those two requests (`/weather`, `/weathers` and `/future_weathers`) out into the function `doWeatherRequest` which takes except for the GET parameter a boolean if it is a current or a future weather data request.

The `/stations` route takes `ip` as a parameter and looks up with a `/ip` request what location the IP corresponds. In combination with the latitude/longitude of the response and the `/location` request, the nearest train stations can be found. After that, a `/stationboard` request with the parameters `id=STATION_ID_OF_RESPONSE` and `limit=5` is enough to get the next train connections running from the nearest train station of this IP location. However, the `limit=5` parameter can respond more than 5 elements if the train leaves on the same time. Therefore, the an additional filter `first(5)` on the JSON response must be applied.

The `/weathers` request takes `ip` as a parameter and forwards the IP to the `/stations` request in order to get the next train connections running from the nearest train station of this IP location. With this response, the latitude/longitude coordinates of these locations can be extracted and forwarded to the `doWeatherRequest` function to get the corresponding weather data. This route responses a list of paris .

The `/future_weathers` route is similar to the `/weather` route except for the parameter `x` which is a mandatory parameter and describes how many days in the future the weather should be responded. As the external weather API does not offer such kind of search criteria, we had to exctract the date of the future weather data and filter that according to the `x` parameter (DateTime.now <= weather_date <= DateTime.now + x-days).

In addition, the routes `/weathers` and `/future_weathers` support sorting. The data can be sorted by adding the optional parameter `sort` with one of the following values: `temperature`, `humidity`, `pressure`, `wind`, `cloud`. If no sort parameter is present, the data is sorted by the temperature.

Error handling

As web service requests mostly have required and optional parameters, we decided to implement an user friendly error

handling by sending an error message to the user with an explanation what went wrong. As this procedure is used in almost all web service requests, an additional function named `doError` is defined which creates the corresponding JSON error response. Because most functions need error handling, we define two helper functions `hasAllParams` and `hasOneParam` to check if a request has the designated parameters.

GUI-client

After the implementation of the CCS-REST application with connecting the RESTful-APIs, implementing the GUI of the WIS-Client application was pretty ici. Thanks to the Sinatra framework you just have to define the HTTP routes within the `main.rb` file. For each defined route (e.g. `/weathers`) you can then define a template view which should be shown to the user (e.g. `slim :index`). Sinatra is compatible with several template frameworks/languages. Some supported frameworks/languages are `erb` (included in Ruby), `haml` or `sass/less`. For this project we used Slim.

For the look of the GUI, we decided to leave the tabs as they were provided in the project template. Since the project template had no starting (index) route initially, we decided to implement a `/home` tab containing a table, which shows the nearest train station for the location found by the users IP and the next 5 destinations sorted by temperature. The Slim template for this index page consists of just 19 lines of code and even handles an empty destinations list properly (e.g. when the used API does not respond). You can see the implementation here: home.slim.

SLIM

The main advantage of Slim is that it reduces the syntax to the essential parts without becoming cryptic. Furthermore, you can implement helper functions for GUI (HTML) elements if you use them often (e.g. an `input` field for a `form`). For our implementation we created a helper function for defining a dropdown-input (`select`) for a `form`. Thanks to this helper it is pretty easy to achieve a dropdown in a template view:

```
# Parameters: friendly_name, name, dropdown_elements, has_empty_element, default_element=-1 (optional, default: nothi
== dropdown_input("Sort by", "sort", ["temperature", "humidity", "pressure", "cloud", "wind"], true)
```

> You can see the full implementation of the helper function here: slim_helpers.rb#L12

Sometimes you cannot achieve everything you want by using plain Slim code. But luckily, you can also combine HTML code with Slim code. As an example, we combined HTML and Slim code in order to implement our `/home` template:

```
- @destinations.each do |destination|
  tr
    td.destination = destination["destination"]["name"].to_s
    | <td><img src="http://openweathermap.org/img/w/#{destination["weather"]["weather"][0]["icon"].to_s}.png">   #{
    td.departure = destination["departure"].to_s
    td.platform = destination["platform"].to_s
```

Conclusion

Although the architecture overview picture seems to look very complex, it was easy to implement it thanks to the choosen technologies and frameworks. Ruby is an easy-to-learn programming language (compared to others) and the frameworks Sinatra and Slim let you focus on the actual implementation and they help you to keep the lines of code low.

Furthermore, the deployment with both Heroku (automatic deploy from commits to master branch) but also Capistrano was very convenient and straight forward.