Meetup Genie Demo

This repository contains a small frontend/server setup, prepared for the Julia Meetup in Copenhagen on the 08.02.2023. The presentation given during the Meetup can be found here.

We will create a simple webpage, where authors can submit a text written for a given project by file upload. We want to store the submission together with a word count.

The following concepts will be demonstrated:

- Creating a new Genie project
- Creating a new React project
- Building a webpage with dropdowns, textbox and file upload using Ant Design components.
- Populating the dropdowns with data stored in a database.
- Receiving the data, analysing and storing it.
- Creating a tabular view into the submitted data.

As an advanced bonus section, we will also show how a websocket connection can be used as a two-way communication between client and server.

Info

Is is advisible to not only look at the final code, but also at the history contained in the git repository. Certain "Checkpoints" will be mentioned in this document, which correspond to tags in the repository. By checking out the tags and only looking at this specific code, it should be easier to follow along the different steps, and trace the changes needed to establish a certain functionality.

The GitHub markdown renderer does not support all features of this document. Download and view the `README.html` or `README.pdf` file for correct display.

Prerequisites

Julia

Install the latest version of Julia.

Open Julia and type

```julia
cd("[path to the directory of this README]")
using Pkg
Pkg.activate(".")
Pkg.instantiate()
```

Node.js

Install Node.js if necessary.

After cloning the repository, open a command line and type

```
cd [path to directory of this README]
cd frontend
npm install
```

to install all the dependencies.

Running the Code

To run the code from any commit of the repository, use the following commands:

Julia

```
cd("[path to the directory of this README]")
using Pkg
Pkg.activate(".")

using Genie
Genie.loadapp()
up()
```

Open a browser and navigate to http://127.0.0.1:8000/.

Node.js Development and Compilation

Open a command line and type.

```
cd [path to directory of this README]
cd frontend
npm start
```

The browser will automatically open http://localhost:3000/ to show the page. This is the development version of the project - any changes to the source files will be reflected in the page shown in the browser.

In order to use the project in production on the server, the source files need to be compiled. This can be done by running

```
cd [path to directory of this README]
cd frontend
npm run build
```

in the command line.

Starting from Scratch

Info

The following steps are not necessary when using the Git repository. They describe how to start completely from scratch.

Creating an Empty Genie app

Open Julia, and make sure that Genie is installed:

```julia
using Pkg
Pkg.add("Genie")
```

To create a new Genie app, use the following commands:

```julia
using Genie
Genie.Generator.newapp_mvc("MeetupGenieDemo")
```

During the process, you will be asked which database to use. For this demonstration, select SQLite as database.

Info

SQLite is a file-based database. Use for example SQLiteStudio for looking at the content of the database-file, and follow along the changes.

Checkpoint

Tag: `1-newapp`

Creating new React App

In a terminal in the project folder type

```
npx create-react-app frontend
```

to create a new project. This will download and install the basic file structure in a subfolder `frontend`. Install some dependencies that we need by typing

```bash
cd frontend
npm install antd
npm install --save @ant-design/icons
```

Template files

The files from `create-react-app` cannot be used directly for our project. We add a simple `frontend/src/index.js` file that contains a menu with two entries.

Putting everything together

There are two steps to arrive at a working setup:

- We adjust the build script to copy the build files to the correct location in the Genie app.

- We change the `routes.jl`-file to respond with the `index.html`-file from the build script (instead of the standard welcome page):

  ```
  route("/") do
    serve_static_file("index.html")
  end
  ```

Checkpoint

Tag: `2-includefrontend`

Getting Started

This tag is a good starting point for exploring the code. Follow the instructions in the section "Running the Code" to start the Genie server, start the Node.js development server, or run the build script to compile the frontend files.

Also have a look at the file structure and the different subfolders and files.

Initial Database and Model Setup

With a new databse, the migration table (a sort of version control for database setups) must be prepared by

```
using SearchLight
SearchLight.Migration.init()
```

This creates a new table `schema_migrations` that tracks changes applied through the Genie app.

We can then start to define our "models" (the data we want to store) - we would like a "User"-model to store information about the users of our frontend, a "Project"-model for project information, and a "Submission"-model to save the submitted texts.. We create all of them with

```
SearchLight.Generator.newresource("user")
SearchLight.Generator.newresource("project")
SearchLight.Generator.newresource("submission")
```

Now we need to define the data we want to store in each model.

User Model

We create a struct that stores the name, e-mail-address and nationality of the users of our frontend in the file `./app/resources/users/Users.jl`:

```
@kwdef mutable struct User <: AbstractModel
  id::DbId = DbId()
  name::String = ""
```

```
  email::String = ""
  nationality::String = ""
end
```

Info

Make sure to add default values to all fields (except the internal database id).

To add a corresponding table to the database, we edit the file `./db/migrations/2022100812460638_create_table_users.jl`. The up function adds the necessary structures, whereas the down function removes them again.

```
function up()
  create_table(:users) do
    [
      pk()
      column(:name, :string, limit=100)
      column(:email, :string, limit=100)
      column(:nationality, :string, limit=20)
    ]
  end
  add_index(:users, :name)
end

function down()
  drop_table(:users)
end
```

Info

When doing the example from scratch, the timestamp in the filename will differ from what is mentioned here.

Project Model

Similarly, we create a struct for storing project information in `./app/resources/users/Projects.jl`

```
@kwdef mutable struct Project <: AbstractModel
  id::DbId = DbId()
  name::String = ""
  description::String = ""
end
```

The corresponding database scheme in `./db/migrations/2022100812461736_create_table_projects.jl` looks like

```
function up()
  create_table(:projects) do
    [
      pk()
      column(:name, :string, limit=100)
      column(:desciption, :string, limit=1000)
    ]
  end
  add_index(:projects, :name)
end

function down()
  drop_table(:projects)
end
```

Submissions Model

The model for storing the submissions from the frontend is prepared similarly to the other models.

The file `./app/resources/submissions/Submissions.jl` contains the definition

```
@kwdef mutable struct Submission <: AbstractModel
  id::DbId = DbId()
  userid::Int = 0
  projectid::Int = 0
  submissiontext::String = ""
  wordcount::Int = 0
  comment::String = ""
end
```

and the file `./db/migrations/2022100813341736_create_table_submissions.jl` defines the database structure

```
function up()
  create_table(:submissions) do
    [
      pk()
      column(:submissiontext, :string)
      column(:userid, :integer)
      column(:projectid, :integer)
      column(:wordcount, :integer)
      column(:comment, :string)
    ]
  end
end

function down()
  drop_table(:submissions)
end
```

Update the Database

Finally, create the database tables by running all the migration files

```
using SearchLight
SearchLight.Migrations.all_up!!()
```

Create and store some user data by running

```
using Main.UserApp.Users
User(name="Holger Danske", email="hdanske@danmark.dk", nationality="DK") |> sa
User(name="Erika Mustermann", email="emuster@de.de", nationality="DE") |> save
User(name="Kari Nordmann", email="kari@noreg.no", nationality="NO") |> save
```

Finally, create some project data by running

```
using Main.UserApp.Projects
Project(name="Intro to React", description="Introduction to reactive programmi
Project(name="Intro to Genie", description="Learning to use the Genie framewor
Project(name="Julia Meetups", description="Reports about the latest Julia meet
```

Info

A proper way to initialize the database with data would be by using "seeds" - see the Genie documentation for details.

Checkpoint

Tag: `3-databasesetup`

Add and Populate Dropdowns

The next step is to add a screen with dropdown menus for selecting the user and project to the frontend, and to fill the dropdowns in with the actual user and project data. To this end, the browser sends a request to the server, which replies with the data from the database.

Frontend Code

In order to allow for communication between the Genie server and the development version of the frontend, we need to add Genie as a proxy by adding a line

```
  "proxy": "http://127.0.0.1:8000",
```

to the beginning of the `./frontend/package.json`-file.

We create a new file `./frontend/src/components/submittext.js` which contains two dropdown menus (one for the user, and one for the project), and a comment textbox. The dropdowns are populated by querying the server for the users and projects:

```
async getData() {
    const data = await fetch(this.props.api + "/getdata");
    const response = await data.json();
    this.setState({
        needsupdate: false,
        datastore: {
            Projects: response.Projects,
            Users: response.Users,
        }
    });
}
```

The data is then used in the dropdowns in the form

```
<Select {...this.formItemLayout}>
    {this.state.datastore.Users.map((dict) => (
        <Select.Option key={dict.id} value={dict.value}>
            {dict.value}
        </Select.Option>
    ))}
    ;
</Select>
```

When finished filling out the form, the form data is send back to the server.

```
onFinish = (values) => {
    fetch(this.props.api + "/formsubmit", {
        method: "POST",
        headers: {
            "Content-Type": "application/json"
        },
        body: JSON.stringify({
            ...values.TS,
        })
    }).then((res) => {
        if (!res.ok) {
            this.declined();
        } else {
            this.success();
        }
    });
};
```

Every form component has a name that starts with TS (e.g. `name={["TS", "User"]}`), and then `...values.TS` collects all entries into a dictionary.

Routes

We need two routes on the server - one for sending the data for the dropdowns, and one for receiving the submitted data.

To populate the dropdowns, we send the data as a json dictionary in the format expected by the frontend. Therefore, we add a function `dropdown_transform` to a new module `./lib/HelperFunctions.jl` for the conversion from a Julia struct to a dictionary with keys `id` and `value`. The route itself looks like

```
route("/api/v1/submittext/getdata") do
  users = all(User)
  projects = all(Project)
  data = Dict("Users" => dropdown_transform(users),
    "Projects" => dropdown_transform(projects))
  return json(data)
end
```

Note how the keys from `data` reappear in the `getData()` function above as `response.Users` and `response.Projects`.

For receiving and storing the submitted data, we add a route that receives data in json format. The `jsonpayload` function spits the data out as `Dict{String, Any}("Comment" => "First Draft", "User" => 2, "Project" => 1)`. From there, we can generate a `Submission` and store it in the database.

```
route("/api/v1/submittext/formsubmit", method=POST) do
  data = jsonpayload()
  comment = data["Comment"]
  user = data["User"]
  project = data["Project"]
  wordcount = 0  # add later
  text = ""  # add later
  Submission(userid=user, projectid=project, submissiontext=text,
    wordcount=wordcount) |> save
  return nothing
end
```

Checkpoint

Tag: `4-basicform`

File transfer

File upload uses an `<Upload />`-component with certain settings:

```
propsUpload = {
    name: ["ST", "file"],
    multiple: false,
    action: this.props.api + "/filesubmit",
    method: "POST",
    onChange: this.onFileUpload,
};
 onFileUpload = (info) => {
    const { status } = info.file;
    if (status !== "uploading") {
        this.setState({ Files: info.fileList });
    }
    if (status === "done") {
        this.setState({
            Wordcount: info.file.response.Wordcount,
            Submissiontext: info.file.response.Submissiontext
        });
    } else if (status === "error") {
        // show error
    }
};
```

The `status === "done"` part is executed when the server responds after receiving the file.

We create a route in the Genie app to handle the incoming file. The content can be accessed via

```
payload = filespayload()
filename = payload["ST,file"].name
content = payload["ST,file"].data
```

Remember that the communication between the server and the frontend is stateless - therefore we analyse the file content and send all information back that are needed for a later submission - including both a word count and the actual file content.

```
result = Dict("Wordcount" => wordcount, "Submissiontext" => text)
return json(result)
```

The full route looks like

```
route("/api/v1/submittext/filesubmit", method=POST) do
  payload = filespayload()
  filename = payload["ST,file"].name
  text = String(payload["ST,file"].data)
  words = split(text, (' ', '\n', '\t', '-', '.', ',', ':', '_', '"', ';', '!'
    keepempty=false)
  wordcount = length(words)
  result = Dict("Wordcount" => wordcount, "Submissiontext" => text)
  return json(result)
end
```

Checkpoint

Tag: `5-fileupload`

Table for Showing Data

In order to show the submitted data, we add a table to the frontend. We use the `Table` component from Ant Design, however we tweak it a little bit to allow for filtering and sorting of entries. This happens in the file `./frontend/src/components/filtertable.js`. The data is stored in `this.state.dataSource`, and the column names are contained in `this.state.columns`. Once the data is received from the server, the `updateColumns` function adds filtering and sorting functionality via the functions `filterData` and `makeSorter`.

As for the server, we need to add a route to get the data from the database and send it to the frontend. This should happen in a format that the frontend can understand. To this end, we have `get_columns` and `get_datasource` in `./src/HelperFunctions.jl` that transform the data. How it must be transformed is based on the documentation of the `Table` component, see here.

The final route:

```
route("/api/v1/view/getsubmissions") do
  users = all(User)
  projects = all(Project)
  submissions = all(Submission)
  columns = ["User", "Project", "Comment", "Wordcount", "Text"]

  table = Dict("dataSource" => get_datasource(submissions, users, projects),
    "columns" => get_columns(columns))
  return json(table)
end
```

Checkpoint

Tag: `6-datadisplay`

Bonus: Websockets

Steep Slope Ahead

This is an advanced topic and usually not needed for simple data display applications.

Normally, the communication between the server and the frontend only goes one way - the frontend sends a request, and the server answers. The server cannot by itself "contact" the frontend to update some data.

If such a functionality is needed, a websocket connection can be established, where the server is able to push messages to all connected clients.

We create a simple component where the user can enter and submit a short message. This message then appears on all connected clients.

Frontend Setup

The code is contained in the file `./frontend/src/components/websocketdemo.js`. We use a `Ref` (a sort of persistent variable) to store the websocket connection itself, and create variables for storing the connection data:

```
const endpoint = "ws_meetup"
const ws = useRef();
const [genieSettings, setGenieSettings] = useState({
  server_host: "1.1.1.1",
  server_port: 8000
});
const [wsOpen, setWSOpen] = useState(false);
```

The `endpoint` is the identifier that the frontend and server use for communication - the server might send data to different endpoints, i.e. different groups of connected clients.

We then define a number of helper functions:

```
const getSettings = async () => {
  const data = await fetch("/api/v1/getwssettings");
  const response = await data.json();
  setGenieSettings(response);
};

const send = (message, payload) => {
  ws.current.send(JSON.stringify({
    'channel': endpoint,   // base URL
    'message': message, // second part after /
    'payload': payload // exposed as payload in Genie
  }));
};

const parseMessage = (dict) => {
  Object.keys(dict).map((key) => {
    switch (key) {
      case "message":
        displayMessage(dict.message);
        break;
      case "error":
        displayError(dict.error);
        break;
      default:
        break;
    };
    return undefined
  }
  )
};
```

- `getSettings` fetches the websocket settings from the server - this still needs to be handled over a standard web protocol before the connection can be "upgraded" to a websocket.

- `send` defines a shorthand function for easily sending data over the websocket connection. The `channel` and `message` parameters need to reappear in Genie's `channel` definition, and the `payload` is what actually get's send.
- `parseMessage` is the main function that defines how the frontend reacts to incoming message. It expects a dictionary, and uses the keys of it to call the code defined in the `switch` statement. The values for each key are used for the data that is needed for the corresponding "action".

Finally, we need to make sure that the server is queried for the websocket settings; and that the websocket connection itself is established. This happens in

```
useEffect(() => {
  getSettings();
}, [])

useEffect(() => {
  ws.current = new WebSocket("ws://" + genieSettings.server_host + ":" + genie

  ws.current.onopen = () => {
    send("subscribe", "")
    console.log('Connection opened!');
    setWSOpen(true);
  };

  ws.current.onmessage = (ev) => {
    const message = parseJSON(ev.data);
    parseMessage(message);
  };

  ws.current.onclose = () => {
    // code for reconnecting
  };

  return () => { };
  }, [genieSettings]
  );
```

`useEffect` ensures that the code gets executed either when the component appears, or when the variables given at the end of the call are changed.

Finally, we also need to make sure to close the connection when the browser window is closed etc., to avoid memory leaks by adding and calling

```
const setupBeforeUnloadListener = () => {
  window.addEventListener('beforeunload', (event) => {
    send("unsubscribe", "");
    ws.current.close();
  });
};
setBeforeUnloadListener();
```

The rest of the code is a form to enter and submit a short message, and a placeholder for messages received from the server.

Server Setup

Server-wise, we need to activate the websocket functionality in Genie by adding

```
Genie.config.websockets_server = true
```

to the `routes.jl`-file. We allow subscriptions to our `ws_meetup` endpoint by calling

```
Assets.channels_subscribe("ws_meetup")
```

To make our life easier, we define the following helper functions:

```
function genie_settings()
    return Dict(:server_host => Genie.config.server_host,
        :server_port => Genie.config.server_port)
end
pure_json = Genie.Renderer.Json.JSONParser.json
Genie.WebChannels.broadcast(ch, msg::Dict) = Genie.WebChannels.broadcast(ch, p
```

- `genie_settings` collects the current IP and port of the Genie server.
- `pure_json` takes a dictionary and outputs a string containing it's JSON representation. (The `json` function used in other places in the `routes.jl` file wraps the JSON string into a HTTP header - we don't need that for websockets).
- Finally, we add a dispatch to Genie's `broadcast` function to be able to easily send dictionaries over a websocket.

The settings are send over a standard route:

```
route("/api/v1/getwssettings") do
  return json(genie_settings())
end
```

Finally, we define a channel to receive incoming messages from a websocket connection:

```
channel("ws_meetup/submitmessage") do
  text = params(:payload)
  msg = Dict("message" => text)
  Genie.WebChannels.broadcast("ws_meetup", msg)
end
```

This channel takes the message from the frontend (contained in `params(:payload)`) and sends it to all connected clients.

Info

> To try it out, open the page in multiple tabs, and send a message from one of them.

Checkpoint

Tag: `7-websockets`

Resources

- Genie Documentation
- Introduction to React
- Create-React-App Docs
- Ant Design Components - use this for looking up which components are available, and what data format they expect.
- Ant Design Mobile Components - similar to the previous framework, but with mobile devices as target.
- SQL Tutorial - rather slow paced, but when finished, one has a good grasp of the concepts.
- Database Migrations: Wiki entry, Article from one of the inventors of the concept.

Errata & Deprecations

The code used in the repository is based on React 16 and Ant Design 4.18. Some deprecations occured in the meantime:

- With React 18, the mounting of the overall `App` component happens in a new way. See commit `e40a8f4` and the React documentation.
- Ant Design uses another syntax for menu items - see the upgrade notes.