Repository for the *HiGHmed Infection Control Dashboard* 2.0

pipeline status

test

This software allows for the visualization of disease spreading in hospitals for both bacteria and viruses.

It consists of

1. a web server built on Node.js ("backend"), which handles connections to the data sources and performs computations on it, and
2. a website ("frontend") served by the same Node.js instance for the actual interaction with the data.

See Structure of the project for details.

(Planned) features of the system:

- Mobile usability of the frontend
- Build and execution system potentially supporting other languages like C++, Python, R, etc. for algorithmic parts
- Extensible to future data sources, procedures, and - very importantly - more visualization modules
- Useful documentation and quality assurance via continuous integration (CI) with Gitlab

Limitations

- Most IDs which are used are strings and *may not contain any commas*, as it uses comma-separated ID lists internally (when communicating with the SQL DB).
- The `IgdSqlDatabaseSource` does not reliably reflect time zones as the underlying data base does not take care of it sufficiently and always runs in local time
- This application is *not* engineered for security in any way. It assumes to be executed in a trusted (virtual) network/environment *only*.
- Dates being processed must all lie between the years 1980 and 2100. This is an intentional restriction to make sure no bogus dates are being processed, e.g. Unix Epoch 0 or the SAP placeholder year 9999. The restriction is easy to lift if required: Simply change the relevant bounds in the schemas or remove them entirely in `src/server/data_io/type_declarations/*.json`. Searching for `"value": "1980-01-01T00:00:00Z"` and `"value": "2100-01-01T00:00:00Z"` should do the job. It might be required if anonymization of the sensible medical and health databases introduces such values.

Installation

The software is *intended* to be run on modern Linux systems only. It will *probably* also run on Windows 10 or other major platforms supporting Node.js & Typescript. Components wich might cause issues include the foreign language libraries. See src/server/foreign_libraries/README.md for details.

The web server requires a recent Node.js version. It is designed and known to be working with version 14 (LTS) and will probably work with future versions too. See here for more information on Node.js versions.

Installing Node.js

Managing multiple versions is easy with the Node Version Manager (NVM), so we will use it in this manual. You can also install Node.js differently and then skip this section.

```
# See https://github.com/nvm-sh/nvm#install--update-script on how to
install NVM
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh |
bash

# install version 14
nvm install 14

# in any new shell, do the following to use the correct node version
nvm use 14
# or set it as the default (the first version that is installed will always
become the default, so this is for more advanced usage)
nvm alias default 14
```

Installing the project & dependencies and building it

The actual installation of the main parts is quite straightforward, although the `install` command might take a couple of minutes:

```
cd to/change/into/a/new/directory
git clone git@github.com:highmed/SmICSVisualisierung.git

# change into the project
cd infectioncontrolsystem2
npm install
```

Now everything should be ready to be built and run:

```
# to build all at once
npm run build
```

To build all parts individually, please have a look at the definition of the `build` command that can be found in `package.json`. Alternatively, details are given in the section on technical details.

Important: Afterwards, please follow the instructions in the foreign libraries documentation to set up the parts written in foreign languages.

Usage

Before you start the server, please check the configuration to set the desired ports. If not changed, sensible defaults will be set. In any way, the starting server will print instructions on how to visit the graphical user interface (GUI) via the browser in a colorful box to the terminal it is started from. You should be able to click the URL that is shown and should be directed to your standard browser.

Starting the Server

After the following command, the webserver should start:

```
# run the server
npm run start
```

More

See [Provided NPM scripts and Compilation](#Provided NPM scripts and Compilation) for more useful commands like running in development mode and exercising unit tests.

Configuration

The configuration can be found and modified in `src/server/config.ts`. The HTTP & HTTPS ports can also be set via environment variables.

Technical details

This section contains information on why certain decisions were made and how stuff works under the hood.

Provided NPM scripts and Compilation

These commands are defined in the `package.json` file. You can easily define new ones, just remember to document them below. They can be used with `npm run <COMMAND>`.

Installation

There are currently no special commands needed for this.

Build process

command	description
`build`	This builds both the server and the client, concurrently.
`build-client`	This builds the client (i.e. the website/frontend) using webpack. It does stuff like Javascript transpilation for compatibility and performance (using Babel & Webpack), reducing the file count by combining stuff like all JS or CSS code from different files and obfuscation.
`build-server`	This builds the server (i.e. the backend/webserver code). This does two things: (1) it invokes `tsc` to build the Typescript code according to `tsconfig.json` and (2) it invokes `build-types`.

command	description
`build-types`	This command constructs Typescript type annotations from the JSON schemes in `src/server/data_io/type_declarations/` to typing information in `src/server/data_io/type_declarations/generated/`. It automatically deletes typing files for deleted schemes. This has to be called each time a schema is modified, removed, or added. However, this is already done each time by `build-server` and thus also by `build`.

Development Mode

command	description
`dev`	This starts both the client and the server in development mode.
`dev-client`	This runs the frontend in development mode. This makes the browser hot-reload on changes to e.g. style and code and restarts the website seamlessly in-place if required.
`dev-server`	This runs the server in development mode. This makes the webserver (backend) restart if server code changes.

Starting Server & Tests

command	description
`start`	This starts the webserver. See also Starting the Server.
`test`	This runs all tests. See also Starting the Server.

Miscellaneous

command	description
`mirror-release`	This "mirrors" the compiled frontend and backend as source code to the Infection Control System 2 - Release repository.

Design Decisions: The Choice of Programming Languages

There are two major parts of this project: the frontend and the backend.

The **frontend** is written with very standard web technologies built around HTML and React. Code is written in Javascript/ECMAScript and then transpiled to a version of it that is supported in sufficiently many browsers. This is a very standard approach. Styling is done in SCSS (that is the "SCSS" syntax variant of the SASS styling language) which is then compiled to CSS. It was chosen because it is a strict superset of CSS and thus allows novice users to also use plain CSS to get started. It is, despite offering more flexibility than CSS, easy to use, and easy to compile with webpack.

The **backend** is written in Typescript, which is a typed "variant" of Javascript. It eventually compiles to just Javascript and enjoys lots of tooling support. It was chosen to provide more (type) safety than Javascript for the many data interfaces. Typescript was chosen above other languages (that might have with better type system or language design) as it still allows us to use all the normal web technologies/libraries around Node.js & NPM. JSON Schema was adopted as a verification technique for the data exchanges as most data is represented as JSON, and it is a technology in widespread use. The platform Node.js was chosen due to its popularity and huge ecosystem of libraries.

Structure of the project

The following diagram visualizes the rough data flow in the project:

```
                      +-----------------------------------------------+
                      |                                               |
                      |               Browser Frontend                |
                      |                                               |
                      +-----+-^----------------------+-^--------------+
npm run start +-+           | |                      | |
                |     +-----v-+-----+ +--------------v-+--------------+
                |     |             | |                               |
+---------------v-+   | Browser GUI | |          Data Access          |
|                 |   | via HTTP(S) | |  via Websockets (via HTTP(S)) |
| start_server.ts |   | src/public/ | |  src/server/websocket_api.ts  |
|                 |   |             | |                               |
+---------------+-+   +-----------^-+ +-^---------+-^-----------------+
                |                 |     |         | |
                |  initialize     |     |         v +
                +-----------------+-----+         ...
```

Therefore, the `src/` directory is divided into the `src/public/` folder with the frontend and the `src/server/` folder with the backend.

Structure of the frontend

TODO: @Tom

Structure of the backend

The following schema gives an overview over the data access via websockets:

```
+-------------------------------+
|                               |
|        Browser Frontend       |
|   or in theory other systems  |
|                               |
+--------------+-^--------------+
               | |
               | | Websockets (via HTTP(S))
               | |
+--------------v-+--------------+ MySQL Remote Queries +-------------------
---+
|                               <----------------------->  SQL DB Data
```

```
Source  |
|          Data Access          <----------+           +-------------------
---+
|  src/server/websocket_api.ts  |          |HTTP(S)
|                               |          |(JSON)     +-------------------
---+
+---------^---------------------+          +----------->     MAH REST API
|
          |                                            +-------------------
---+
+---------v--------+
|  GetDataSources  |
+------------------+
```

If something else than the special `GetDataSources` procedure is called, the following steps are performed (error handling aborts immediately):

```
+-------------------------------+
|                               |
|        Browser Frontend       |
|   or in theory other systems  |
|                               |
+--------------+-^--------------+
               | | Websockets (via HTTP(S))
           (1) | | (10)
               | |
+--------------v-+---------------+ (2) +---------------------+
|                                +----->  resolveDataSource  |
|          Data Access           |     ++--------------------+
|  src/server/websocket_api.ts   | (3)  |
|                                <------+
|                                |
|                                |     +-----------------------------------
--+
|                                |     |
|
|                                |     |   Some Concrete Data Source
|
|                                |     |
|
|                                | (4) |   +-----------------------------+
|
|                                +--------->     Validate Parameters     |
|
|                                |     |   +-----------+-----------------+
|
|                                |     |               | (5)
|
|                                |     |   +-----------v-----------------+
| (6) +---------------------------+
|                                |     |   |     Transform Arguments     +-
```

```
-------->                            |
|                                 |     |   +------------------------------+
|     |                            |
|                                 |     |
|     |  Remote Database or API    |
|                                 |     |   +------------------------------+
| (7) |                            |
|                                 |     |   |     Transform Response      <-
--------+                            |
|                                 |     |   +-----------+------------------+
|     +----------------------------+
|                                 |     |               | (8)
|
|                                 | (9) |   +-----------v------------------+
|
|                                 <---------+     Validate Response       |
|
|                                 |     |   +------------------------------+
|
|                                 |     |
|
+---------------------------------+     +-----------------------------------
--+
```

How to extend this software

This section contains tutorials on how to extend various components of the system.

Changing or Adding new JSON Schemas

Validation is performed using JSON Schemas. Information on how to use JSON schemas can be found in the tutorial on the official website. We use ajv for the general validation and the plugin ajv-moment for validation of dates. Please referer to the libraries for documentation.

To add a new schema called `TheSchemaName`, two things must be done: (1) the actual schema must be placed in `src/server/data_io/type_declarations/TheSchemaName.json` and (2) the schema must be referenced for ease of use. Referencing makes the schema available in the function `ensureIsValid()`. The schema name is the ID given within the schema (and not determined by the file name, although please keep them the same for the sake of sanity!).

(1) When creating the JSON file of a new schema, copying an existing one from `src/server/data_io/type_declarations/` is probably a good starting point. Make sure to modify the `"$id"` attribute (probably right at the start of the schema). Now it's probably good to execute `npm run build-types` in order to generate an interface in `src/server/data_io/type_declarations/generated/TheSchemaName.d.ts` (there is no need to remember that specific file).

(2) Referencing the schema can be done in `src/server/data_io/types.ts`. You can mainly follow the code for the existing schemas for procedure arguments and result data. The detailed steps are:

1. Import the actual JSON schema by adding `import * as TheSchemaName_JSON from "./type_declarations/TheSchemaName.json"` to the top of the file.
2. Add the `TheSchemaName_JSON` to either `ARGUMENT_SCHEMAS` or `DATA_SCHEMAS`. Alternatively, you can also create a new group with your new schema and add the new group to the `Ajv.Options.schemas` below.
3. Import the interface/typing information by adding `import {TheSchemaName} from "./type_declarations/generated/TheSchemaName"` to the top of the file.
4. Export the type similar to the other ones. This makes you new type `TheSchemaName` importable from the file `src/server/data_io/types.ts`, where all other declarations live.

Adding new data sources

All data sources must extend the abstract class `AbstractDataSource` (found in `src/server/data_io/abstract_data_provider.ts`) and are usually placed in `src/server/data_io/concrete_data_providers/`.

The documentation of `AbstractDataSource` describes how the class should roughly behave. It should suffice to implement the abstract data retrieval methods. As part of that, you might have a look at existing data sources.

In the end, make sure to add tests to `test/test_all_data_sources.ts` by extending `PROCEDURES`.

Adding new procedures

New procedures can be added by first modifying the `AbstractDataSource` in `src/server/data_io/abstract_data_provider.ts`. Here, add a method to the end of the class, analogous to the other methods: `public abstract async Name_Of_The_Procedure(parameters: Arguments_Whatever): Promise<ResultingDataType>`. Also add an entry to `AbstractDataSource::MAPPING` to allow for dynamic calls via `AbstractDataSource::callByName`. Finally, the actual procedure must be implemented in all data sources, which may explicitly throw and error if they shall not or cannot support it.

In the end, make sure to add tests to `test/test_all_data_sources.ts` by extending `PROCEDURES`.

Documentation of the data sources

TODO: add link

See also

- The archetypes can be found like this: visit the *HiGHmed* Clinical Knowledge manager -> select "Use Case Infektionskontrolle" at the top as "project" -> select the orange tab "Templates" on the left -> expand "EHR Templates -> Composition"
- GitLab repository of queries

Questions / TODOs

- Personaldaten-Abfrage -> noch nicht modelliert in CKM, Vorhaben das als "Demographie"-Archetypen zu modellieren, aber eigentlich Trennen der Demographie von Klinischem, Verknüpfung über ID;

abwarten aber dann vermutlich möglich, das gibt es auch eine rechtliche Diskussion, an manchen standorten ggf. nie Daten drin, weil sie strikt getrennt sind

- Stronger validation, i.e. sorted-ness, ...
- Linter?
- deploy (started in CI file)
- License & license policy
- Labordaten validierung: wenn befund da => dann KeimID auch da