



CHARLES --- EXPLORER

Developer documentation

Bc. Jindřich Bär

Department of Software Engineering
Charles University
Prague, Czech Republic

Contents

1	Introduction	2
2	Technologies used	4
2.1	Frontend	4
2.2	Backend	4
2.2.1	Database	4
2.2.2	HTTP Server	4
2.2.3	Search engine	4
3	Project architecture	5
3.1	Docker setup	5
3.1.1	Core services	5
3.1.2	Auxiliary services	6
3.2	Packages	6
3.2.1	packages/webapp	7
3.2.2	packages/nanoker	8
3.2.3	packages/feeder	8
3.2.4	packages/solr-client	8
3.2.5	packages/prisma	9
4	Deployment	10
4.1	Prerequisites	10
4.2	Installation	10
4.3	Data integration	12
4.3.1	Prerequisites	12
4.3.2	Data transformations	12
5	Amendment - explorer.cuni.cz	15
5.1	Accessing the server	15
5.2	Server	15

1 Introduction



Here follows a part of the project specification from the initial project proposal (Záměr firemního projektu). Changes that were made to the final version of the project are denoted by footnotes.

Information about courses, teachers, scientific publications or study programmes at Charles University are provided mainly by the Study information system (<https://sis.cuni.cz>).

As a result of long-term development and continuous maintenance of the information system, the access to data in the web application is divided into several separate modules, which makes it very difficult to browse and view the data in context. The data is also not conveniently indexed and the system provides only limited search capability.

The Charles Explorer project addresses these issues by simplifying the data model and by creating well-defined data schemas that clearly correspond to real-world world. In addition, the project will allow data to be searched using advanced full-text search. Finally, the project will also offer the user a graphical interface for exploring the data in context.

In implementing the project, the student will learn about the actual structure of the data in information systems of Charles University. The student will design a suitable scheme for managing data on individual entities of the university (lecturers, courses, study programs, publications, etc.) and the relationships between them. Then the student will design and implement an automatic pipeline to process the data from its current form into a new schema.

Next, the student will implement the backend and frontend of a web application that will allow the user through a user-friendly interface over the data to perform full-text queries and browse and view entities at the university in context.

The user experience for searching should match that of conventional commercial search engines (e.g. automatic suggestions, **typo correction**¹ etc.). During this phase of the project, the student will become familiar with, among other things, linguistic text analysis tools.

¹While the search engine does fix small typos in the queries internally to improve search recall, the system does not have the "Did you mean" feature - i.e. suggesting the typo fix - yet.

Finally, the student will create **automatic tests for the application**² and deploy the pipeline to university server.

The output of the project in the form of an application will be used as presentation material for the university, for new applicants or as a navigational aid for existing students.

The project will also be used by the different parts of the university in the search opportunities for cross-faculty collaboration or in finding suitable tutors for new courses.

²At the current state, the application has only been tested using Grafana k6 (a load testing tool) and is missing any more extensive tests.

2 Technologies used

The original plan - as specified in the project proposal - was to use the following technologies:

- **Frontend:** React, Typescript
- **Backend:** Solr, MySQL, Express.js, Typescript

Since the original project proposal, the following changes have been made:

2.1 Frontend

The (vanilla) React framework was replaced with **Remix.js**. Remix is a full-stack Node.js framework built on top of React and is designed with progressive enhancement in mind.

It also provides a routing solution and a data fetching solution which helps to keep the codebase clean and consistent.

2.2 Backend

Several changes have been made to the backend part of the project:

2.2.1 Database

The current version of Charles Explorer is using **PostgreSQL** to store data and **PrismaORM** for retrieval and management. This change has been made mostly because PostgreSQL provides better developer experience than MySQL when working with PrismaORM - most notably, MySQL required us to provide maximum text length for every string field.

PostgreSQL also provides wider variety of structured data types (array types, XML), which might come in handy later.

2.2.2 HTTP Server

As mentioned above, the entire project has been migrated to Remix.js, which comes with a bundled server solution. This way, Remix.js can take care of both frontend and backend, solving the server - client data transfer logic for us.

2.2.3 Search engine

The project does use **Apache Solr** as the internal search engine.

3 Project architecture

The project is tracked on GitLab and is available at <https://gitlab.mff.cuni.cz/barj/charles-explorer>.

Structured as a monorepo, all parts of the project are contained in the same repository in the `packages` folder. Furthermore, the root folder contains project-wide configuration files for the ESLint linter (`eslinttrc.js`), the Yarn package manager dependency declarations (`package.json`, `yarn.lock`) and the `tsc` transpiler configuration (`tsconfig.json`). It also contains a sample SQLite3 database for seeding the application (`explorer.db`).

3.1 Docker setup

The core application itself consists of four `docker compose` services, two other services are defined in the root `docker-compose.yml` file.

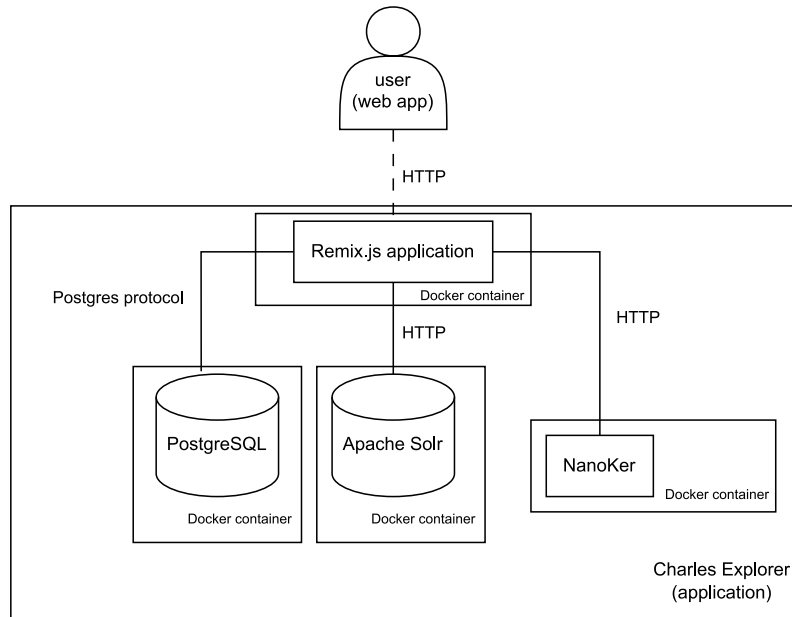


Figure 1: A diagram describing the top-level project architecture

3.1.1 Core services

First is the `web` service, which is the main web application. This container's image is built from the `dockerfiles/Dockerfile.app` file and contains the

production build of the main Remix.js application. This is also the only service that the client is supposed to be communicating with.

The `db` service is built on top of the official `postgres` Docker image and runs this image basically without change. It mounts the `postgres-data` folder from the repository root for data storage, easier debugging and more transparent recovery in case the container shuts down.

The `solr` service is built on top of the official `solr` Docker image. It mounts the `solr-data` folder from the repo root as the index storage. The `solr-data` folder by default contains presets and other files (e.g. stopwords files, text field configurations for different locales etc.).

The `ker` service for generating the keywords for the wordcloud is running a custom Docker image, built from the `packages/nanoker/Dockerfile` file. This package (`packages/nanoker`) is a simplified Python3 port of the Ker application by Mgr. Jindřich Libovický PhD. of ÚFAL, MFF. All credits go to him and the MorphoDiTa team.

3.1.2 Auxiliary services

The root `docker-compose.yml` file contains definitions for two more services.

The `solr-user-fix` is a one-time service, designed for fixing ownership of the aforementioned `solr-data` folder contents. There is a well known issue in the official `solr` Docker image that forbids the Solr from writing into the mounted volume if the `solr` user is not the folder owner.

The `seed-data` service is running a custom Docker image, built from the `dockerfiles/Dockerfile.importer` file. This service connects to the common `docker network` and seeds the Postgres database container and the Solr container with data from the user-modifiable `explorer.db` database file.

3.2 Packages

This subsection describes the individual packages that are part of the Charles Explorer project, their purpose and their dependencies. It should help project maintainers to understand the project structure and help developers to understand the codebase.

3.2.1 packages/webapp

This package contains the main web application. It is built on top of the Remix.js framework and is written entirely in Typescript.

Structured as a Node.js NPM-style package, it contains a `package.json` file with the package metadata and a `tsconfig.json` file with the Typescript compiler configuration. Aside from those it also contains `tailwind.config.js` and `postcss.config.js` files for the Tailwind CSS framework configuration. The Remix.js configuration is located in the `remix.config.js` file.

The `styles` folder contains the global CSS stylesheets - if you want to add a new CSS rule, this is the place to do it. All the other CSS files in the project are automatically generated using Tailwind CSS and should not be modified manually.

The `public` folder contains the static assets (images and localisation files) that are served by the application. In case you want to add a new localisation, you can do so by following the structure of the already existing localisation files (e.g. `public/locales/cs/common.json`).

The `app` folder contains the main application code:

- The `assets` folder contains static assets (SVG files) that are used in the application components. The fact that those are imported directly into the code is a reason why they are not located in the `public` folder.
- The `components` folder contains the React components that are used in the application. The components are structured into folders based on their purpose.
- The `connectors` folder contains modules that are used to interface the application with the backend services (currently Prisma client and a minimalist Solr client implementation).
- The `routes` folder contains the application routes as per the Remix.js specification. A `.tsx` file in this folder represents a single (possibly nested) route in the application.

In other words, the files here combine the components from the `components` folder into pages that are then served by the application. The `index.tsx` file is the entrypoint of the application.

- The `utils` folder contains utility functions (mostly data transformations) that are used throughout the application. There is no structure to this

folder, as it is not expected to grow very large.

- The `i18n.ts` file contains the internationalisation logic. It is used to load the localisation files from the `public` folder and to provide the `t` function that is used to translate the application.

In case you want to add a new localisation, you can register it in the `i18n.ts` file.

- The other files in the `app` folder are either automatically generated or are used for application configuration and shouldn't be relevant for the project maintainers.

3.2.2 packages/nanoker

This package is a simplified Python3 port of the `ker` application by Mgr. Jindřich Libovický PhD. of ÚFAL, MFF. All credits go to him and the MorphoDiTa team.

The package follows the original `Ker` application closely, for any further information please refer to the original documentation.

3.2.3 packages/feeder

The main part of this Node.js NPM package is **Charles Importer**, the tool for importing the data from the SQLite3 database into the PostgreSQL database and the Solr index. The entrypoint of this package is located in the `src/index.ts` file.

Most of the application is written in a very straightforward, procedural manner.

The only specialty here is the `transformers.ts` file, which contains the data transformation functions. These functions are written in Typescript and are used to transform the data from the SQLite3 database into the Charles Explorer data model. The user is expected to modify these functions to match their database schema.

3.2.4 packages/solr-client

This package contains a minimalistic Solr client implementation. It is used by the `webapp` and `feeder` packages to communicate with the Solr instance.

It is written in Typescript and is structured as a Node.js NPM package. The main file (`src/index.ts`) exports the client class.

3.2.5 packages/prisma

This package contains the Prisma client definition and the Prisma schema. It is used by the **webapp** and **feeder** packages to communicate with the PostgreSQL database.

The **schema.prisma** file contains the Prisma schema definition as per the Prisma documentation. The **migrations** folder contains the generated SQL migrations. The actual Prisma client is generated by the **npx prisma generate** command, which is run automatically by the **docker compose** command.

4 Deployment



This part of the documentation is mirrored from the project wiki. While these are identical at the time of writing this documentation, only the wiki version will be updated. Please prefer the wiki version whenever possible.

4.1 Prerequisites

In order to run Charles Explorer locally, you will need to have the following installed:

- Git
- Docker and Docker Compose

Some basic knowledge of Docker and Docker Compose is also required. While it is possible to run the Charles Explorer instance without Docker, we do not provide any instructions for this, as it is not recommended and not supported.

4.2 Installation

After installing the prerequisites, you can start by cloning the Charles Explorer repository:

```
git clone git@gitlab.mff.cuni.cz:barj/charles-explorer.git
```

Navigate to the **charles-explorer** directory and create a **.env** file with the URL of your Solr instance, Nanoker instance and PostgreSQL database. Chances are, you don't have any of these - in that case, you can use the supplied **.env.prod** file - just rename it to **.env**.

Now, you can run the Charles Explorer instance using Docker Compose:

```
docker compose --profile=app up
```

After a while, your Charles Explorer instance should start running on port 8080. However, it won't be very useful yet, as the database doesn't contain the required tables yet. You can see that the underlying services are having a hard time understanding that by looking at the logs of the **docker compose** command.

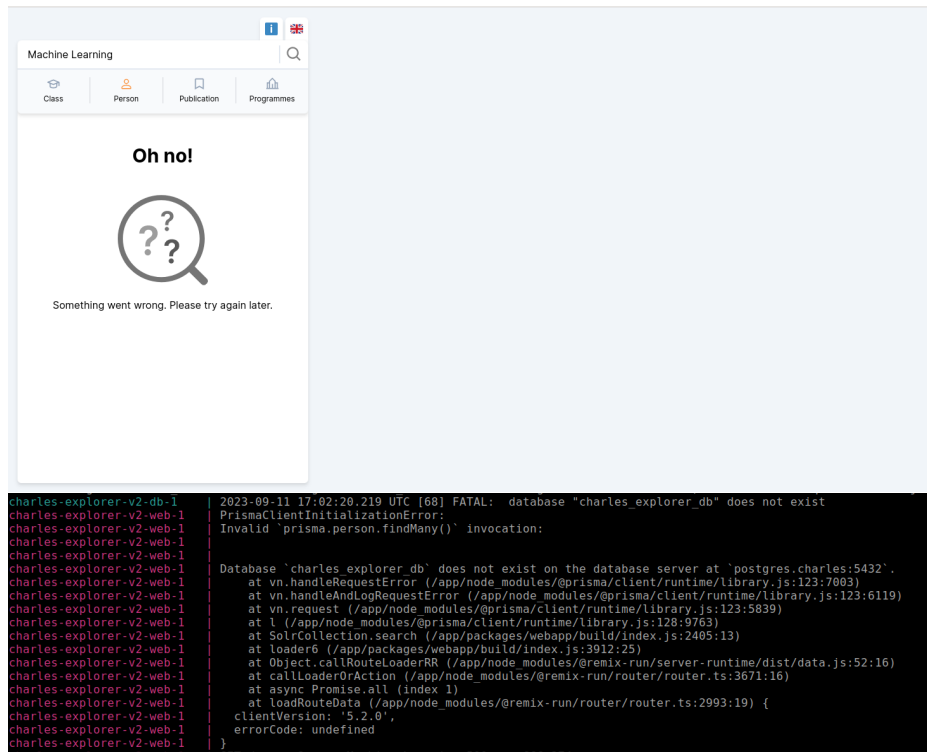


Figure 2: The application is running, but shows an error. Closer look at the backend console tells us that the PostgreSQL database does not exist yet.

While that might look like a big problem, we're actually halfway there - our custom Charles Explorer instance is already live! To fix the data issue, let's continue to the Data integration step.

4.3 Data integration

After successfully deploying the application to our server, we now want to fill it with our data. While this might sound scary, Charles Explorer tooling actually makes it an easy task.

4.3.1 Prerequisites

- Running Charles Explorer instance (if you don't have that, go back to the deployment guide).
- Our data export as a SQLite3 database file.

If you don't have your data packed up yet, don't worry. This repository contains a sample `explorer.db` file with sample data you can use to try out the app.

4.3.2 Data transformations

First, we have to put the database file to a place where Charles Explorer tooling will be able to find it. Take your SQLite3 database file and save it to the root of this repository. The file has to be named `explorer.db` (you will replace an already existing sample file).

You might have noticed we haven't spoken about your database's schema yet. This is because we're fully aware that every university might be storing their data in a different system. To simplify the integration process, Charles Explorer works with a data model that is independent of the actual database schema. This however also means that you will have to map your database schema to the data model.

To do this, you have to edit the `transformers.ts` file in the `feeder` project. This file contains a set of functions that map your database schema to the data model. The functions are written in TypeScript, but you don't have to be a TypeScript expert to understand them. The functions are well documented and you will only have to change the SQL queries to match your database schema.



Tip: Save yourself some time and install a Typescript extension in your IDE. The `transformers.ts` file is loaded with type definitions that will guide you and enable autocomplete and checks.

This way, you will immediately see what fields does which entity have and how to supply them from your database.

```

{
  // ...
  person: {
    query: 'SELECT id, name FROM PEOPLE_IN_MY_DATABASE'
    transformer: (row) => {
      return {
        id: row.id,
        name: {
          create: {
            value: row.name
            lang: "cs"
          }
        }
      }
    }
  },
  publication: {
    //...
  }
}

```

Similar to the codeblock above, the **transformers.ts** file contains a single object with predefined keys (faculty, department, person, class, programme and publication). Each key represents a single entity in the data model. The value of each key is an object with two subfields: **query** (a string containing a SQL query) and **transformer** (a function that transforms the result of the query into the data model).


While this might seem complicated at first, it's actually quite simple. The query field contains a SQL query that will be executed in your SQLite3 database. The transformer field contains a function that will be called for each row returned by the query. The function takes a single argument (row) that contains the row returned by the query. The function then returns a Prisma **create** argument that represents the entity in the Charles Explorer data model.

You can take inspiration from the example queries and transformers in the **transformers.ts** file. If you're not sure how to write a query, you can use a tool like DB Browser for SQLite³ to help you.

³DB Browser for SQLite only supports SQLite 3.12.2 (as of September 19, 2023), while the Charles Explorer tooling bundles SQLite v3.41.1. While all the features of DB Browser will be available in the Charles Explorer import tool, it might not work the other way around.

```
docker compose --profile=seeder up
```

```
charles-explorer-seed-data-1 All migrations have been successfully applied.  
charles-explorer-seed-data-1 npm notice  
charles-explorer-seed-data-1 npm notice New major version of npm available! 9.8.1 -> 10.1.0  
charles-explorer-seed-data-1 npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.1.0  
charles-explorer-seed-data-1 npm notice Run npm install -g npm@10.1.0 to update!  
charles-explorer-seed-data-1 npm notice
```



CHARLES IMPORTER

```
100% | 26/26 | ETA: 0s | Elapsed time: 1s | 🟡 Inserting FACILITY  
100% | 1819/1819 | ETA: 0s | Elapsed time: 2s | 🟢 Inserting DEPARTMENT  
100% | 34038/34038 | ETA: 0s | Elapsed time: 11s | 🟢 Inserting PERSON
```

The Charles Importer tool opens your `explorer.db` database file, runs the transformations and feeds them to the database. After the database seeding is done, it also indexes the records in the attached Solr instance, so the full-text search is working.

After the Importer finishes, your application is ready to go! Go to `http://localhost:8080` and give it a try!

In case you have encountered any issues in the process, please, let us know in the GitLab issues. Thank you!

14

5 Amendment - explorer.cuni.cz



This part of the documentation is solely about the Charles Explorer instance deployed at <https://explorer.cuni.cz>. It describes deployment details that are probably not applicable to any other deployment environments.

The pioneer Charles Explorer instance is available at <https://explorer.cuni.cz>. The system is running on the `nomos.is.cuni.cz` server under `bar` user.

5.1 Accessing the server

While the client side of the application is accessible at <https://explorer.cuni.cz>, the (`ssh`) access to the server is limited only to clients from within the Charles University network.

During development, we've circumvented this limitation by using an SSH jump server. The jump server is a server that is accessible from the outside and from which we can access the internal network. For this purpose, we've used the MFF lab computers at `u-plX.ms.mff.cuni.cz` (where `X` is a number from 0 to 37).

To access the production server, we've used the following command:

```
ssh -J u-pl0.ms.mff.cuni.cz bar@nomos.is.cuni.cz
```

This requires you to have an account on the MFF lab computers (or any other computer used as the jump server) and an SSH key from the lab computer to the production server set up.

5.2 Server

On the production server, the home folder under the `bar` account has the following structure:

- `charles-explorer` - the source code of the application (`git clone` of the GitLab repository). Do not make any changes here manually, as the automatic deploy scripts can overwrite your changes on the next change in the remote repository (see below).
- `sql/sis-to-sqlite3.sh` - a shell script that connects to the (Oracle) SIS

database and exports the data to a SQLite3 database. This script needs to be run in a Docker container with Oracle client installed (see below).

- **update-data.sh** - a shell script that runs the **sis-to-sqlite3.sh** script in a Docker container with the **sql** folder mounted as a volume, and then copies the resulting SQLite3 database to the **charles-explorer** folder. This script can be run from the GitLab CI pipeline.
 - The CI pipeline is configured to run this script only when run manually. This is because the script takes a long time to run and we don't want to run it every time a commit is pushed to the repository.
- **rebuild-web-app.sh** - a shell script that rebuilds the web application. This script is run from the GitLab CI pipeline on every push to the **master** branch. This behaviour can be prevented by adding **[ci skip]** to the last commit message of the push.



Both of the CI-executed scripts are set up as SSH forced commands. This is possible because the hosting GitLab server is in the university network and can connect to the production server via ssh.