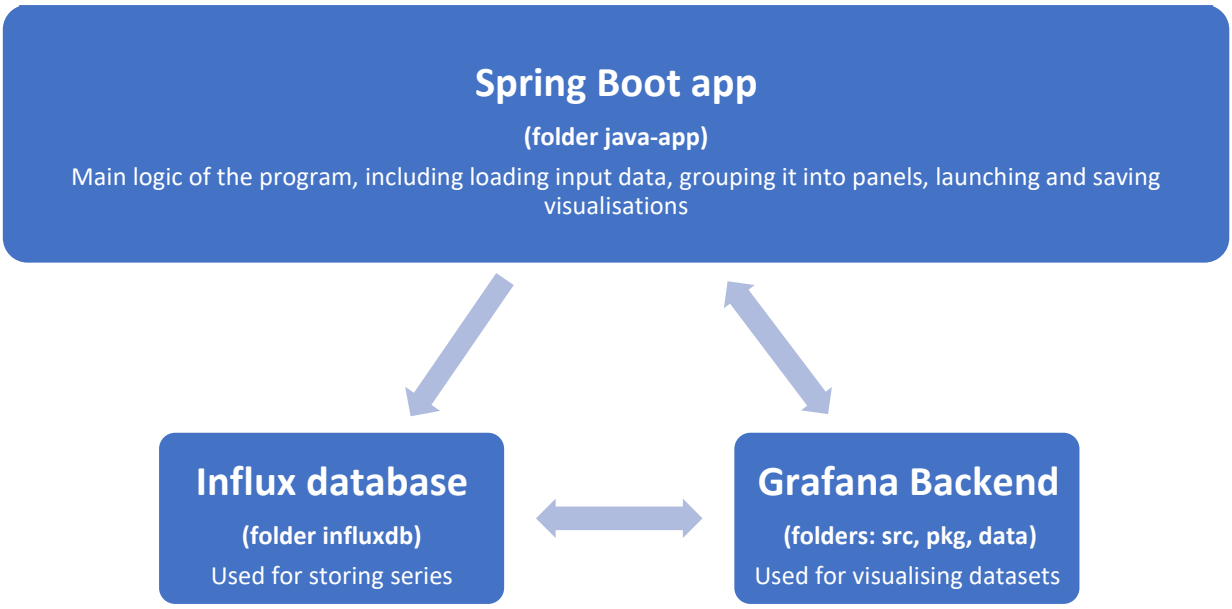
Online Monitoring Tool

Overall structure

The app structure consists of three components:



Other important folders:

- **Configs**

Default.json stores default metadata such as operator, oven etc.

- **Output**

Direction where saved simulations go

- **Sample input**

Stores sample datasets you can use for testing of the monitoring tool

Spring Boot App

Java app has a typical Spring Boot structure:

- **Controllers**

  Mapping to enable backend/frontend communication
  MainController.java is for communication within Spring Boot app itself,
  ExperimentDataRestController.java enables requests from Grafana frontend

- **Repositories**

  SavedSimulationsRepo.java processes saved experiments in /output folder

- **Model**

  Data representation as objects, e.g. Simulation, Panel, Comment

Functionalities

1. **Launching app**
   After `SpringApplication.run` is ready, we must launch Grafana and influx servers.
   Corresponding ports need to be free (if they are already occupied with Grafana/influx process, app will just use the same instance).
   Ports can be customized in application.properties. Each of two processes has to run in a separate thread.
2. **Welcome screen** At **Global variables:**

   @RequestMapping("/home") directories a welcome screen is shown. `readSavedSimulations()` parses saved simulation data, their names are displayed. User can view a saved experiment or start a new one.

3. **Loading a saved simulation**

   At /load?id={simulation id} a saved experiment is launched. Spring Boot app gets the needed experiment by its id and pushes data into database.

   Grafana frontend (src/github.com/grafana/grafana/public/app/custom.ts) makes an ajax request to get experiment data, changes default panel names, renders comments.

4. **Starting new experiment**
   After clicking on “Start new experiment” user is redirected to /newsimulation

   Add panels
   Here user must configure at least one panel. After clicking on an empty panel slot, we are redirected to /newpanel, which opens a popup with panel settings. Here you can enter custom panel name and select datasets for visualisation. After some data was already entered, further changes will be send to /editpanel/{panel id} . Ready panel is posted to /newpanel/{panel id} with parameter “new panel”

   Add files
   After you select a file for upload and click on “Add file”, data is send to mapping /newpanel/{panel id} is called **with parameter “new file”**

Starting visualisation
After you finished configuring new experiment and click submit button, data is posted to /newsimulation

5. **Adding static data**
   Dynamic sensor data is launched at once, static data is added on a button press. Signal is sent to /launchStatics

6. **Writing comments**
   When Grafana is running, you can submit a comment to your experiment. It will be posted to /sendComment

7. **Save simulation**
   Saving is proceeded at /startSavingPanel

Classes and methods

Simutool.controllers

`MainController.java`

Global variables:

```
public static Simulation pendingSimulation;
Simulation that is currently active – saved or new.

static Panel pendingPanel;
Panel that is currently being created or edited.

public static List<Panel> pendingPanels;
List of precreated panels in a pending simulation.

public static boolean staticsAreLaunched = false;
Shows if static datasets were already launched (by default, dynamic sensor dataset is
launched first, simulations and curing cycles are added when a corresponding button
is clicked)

public static boolean experimentStarted = false;
Is set to true, when the user is ready setting up a new simulation and runs it or
clicked a saved one. Thus, pending data can be discarded now.

boolean allPanelsAreStatic = true;
If set true, there was no dynamic sensor dataset selected in any of the panels.

String redirectLink;
Initial Grafana link, to which user is redirected. If user adds/removes any
parameters in Grafana view, he can always restore the initial one.

String refreshingPar;
Defines, to which of three preconfigured dashboards user shall be redirected: one,
two or three panel view.
```

```
public static List<InputJSON> meta;
```

Stores metadata tulips.

Methods:

@RequestMapping("/home")

```
public String startMenu(Model m)
```

Launches reading of saved simulations with `readSavedSimulations()` and renders a welcome screen.

```
@RequestMapping("/load")
public String loadSavedSimulation(@RequestParam("id") String id)
```

Clears influx database first.

Gets simulation by id, it there are any comments, writes it to the database with `writeCommentsToDB()`

Pushes all datasets to influx via `addStaticPoints()`

Calculates perfect scale for visualisation to show data from earliest time point to the latest.

Finally, chooses a redirect link to Grafana depending on the number of panels.

```
@GetMapping("/newsimulation")
public String getSettingsForm(Model m)
```

If called for the first time, when simulation has no pending data yet, set `pendingSimulation` to a new Simulation, clears pending panel (pending data is data that is currently being created/edited and wasn't saved yet).

Reads metadata from default.json file and writes it to `meta` variable.

Start date and time are saved here.

```
@GetMapping("/newpanel")
public String getPanelForm(Model m, @RequestParam(value="simulation", required=false)
String simName)  returns template with a clear Panel form
```

```
@PostMapping("/newsimulation")
public String saveSimulation(@ModelAttribute Simulation simulation, Model m, final
RedirectAttributes redirectAttributes)
```

Shows error message if experiment doesn't have a name or there were no panels added.

`processPendingData()` processes submitted simulation and panel data

```
@PostMapping(value="/newpanel/{panelId}", consumes = "multipart/form-data", params =
"new file")
public String saveFile(@ModelAttribute Panel panel,  @PathVariable(value="panelId")
```

```
Integer id, HttpServletRequest request, Model m, final RedirectAttributes
redirectAttributes)
```

Uploads a new file.
Not to confuse with same mapping and param = "new panel"
Won't accept anything but csv.
Parses file with `parseFilesForPanels()`
If no name was submitted, adds a default one.

```
@PostMapping(value="/newpanel/{panelId}", consumes = "multipart/form-data", params =
"new panel")
public String savePanel(@ModelAttribute Panel panel,  @PathVariable(value="panelId")
Integer id, HttpServletRequest request, Model m, final RedirectAttributes
redirectAttributes)
```

If panel is new, add to pending panels.
Not to confuse with same mapping and param = "new file" If
panel with such id already exists, update.

```
@GetMapping(value="/editpanel/{panelId}")
public String editPanelForm(@ModelAttribute Panel panel,
@RequestParam(value="simulation", required=false) String simName,
@PathVariable(value="panelId") Integer id, Model m, final RedirectAttributes
redirectAttributes)
```

if some panel in pendingPanels already has id that was passed, set it as pendingPanel – panel that is being edited currently.

```
@GetMapping("/savePanel/{exit}")
public String startSavingPanel(@PathVariable(value="exit") boolean exit)  Parameter
exit signalizes whether experiment shall be stopped after saving.
Sets a unique identifier for experiment.
Launches saving at savePanels()
```

`public void processPendingData()`
Iterates over all datasets.

Picks only sensor files, sets internal number within panel (sensor 1, sensor 2 etc.) Sets panel number.

Clear database with `tearDownTables()` and start pushing sensor data with `simulateSensor(int millis, List<FileDTO> sensorData)`.
Redirect to one of three Grafana dashboards (1, 2 and 3 panel view).

`ExperimentDataRestController.java`

```
@PostMapping("/sendComment")
public void sendComment(@RequestBody Comment commentData)
```

Comment data is a timestamp and a comment text.

What we get from frontend is a time in format “hh:mm:ss”, after adding year, month and date we send it to `normalizeTimeStamp`() to get a timestamp.

```
@GetMapping("/launchStatics")
public boolean launchStatics()
```

Iterates over pendingPanels and creates two ArrayLists for simulations and curing cycles.

Push both ArrayLists via addStaticPoints()

Set pendingSimulation is loaded, meaning static data was launched.

Simutool.DBpopulator

`InfluxPopulator.java`

Global variables:

```
Timer timer;
```

Used for scheduling while pushing sensor data.

Methods:

```
public void addStaticPoints(List<FileDTO> simData, String type)
```

Gets a List<FileDTO> with static data.

Finds difference between first timestamp and current moment (shift).

Thus, with this difference added to every timestamp, experiment that was in past starts as if it were "from now".

```
public void tearDownTables()
```

Delete databases for main data and comments and creates them anew. Stops and clears Timer if there is any.

```
public void simulateSensor(int millis, List<FileDTO> sensorData)
```

Every dataset is pushed to the database in a separate thread, one value per certain interval, defined in **millis** parameter.

We put a CyclicBarrier in order to start all threads as simultaneously as possible.

Continuous pushing is implemented as a Timer and a TimerTask, represented by a custom class `AddPoint`.

```
class AddPoint extends TimerTask {

    Timer timer;
    A reference to the Timer needs to be passed.
```

```
int millis;
Interval in milliseconds

int counter;
Index of the line with the next value to push

int panelNum;
Number of panel dataset belongs to

long shift;
Difference between first timestamp and

int internalNum;
```

Internal dataset number within one panel, e.g sensor1, sensor 2

Data is pushed as field `"P1_sensor_3”` where 1 is panel number and 3 – internal number, meaning, there exist at least two more sensors.

Simutool.repos

`SavedSimulationsRepo.java`

Global variables:

`static List<Simulation> savedSimulations`
For keeping saved simulations after parsing from .csv

Methods:

`public void readSavedSimulations()`
Retrieves `savingFolder` from appliaction.properties and iterates over directories there. Creates Simulation objects for every found simulation:

- Name is retrieved from folders name
- Iterate through files in each directory
- If file is comment file – start parsing with `parseFilesForPanels()`
- If file is metadata (internal) – parse with `parseMetadata()`
- For the rest files: retrieves panel name from file name and groups data **(panel name is everything between "PANEL_" and "---")**, creates Panel object for every panel
- For files in each panel group: Retrieves file datatype **(datatype is everything between "---" and file.length()-6)**
- Parses these files with `parseFilesForPanels()`
- Find files with earliest and latest time, save them to show the right scale
-

`public void writeCommentsToDB(FileDTO commentsFile)`
Pushes time-value points to influx table for comments (see application.properties)

Data is a List<String[]> which is taken from FileDTO.getRows(), for every line value at index 0 is a timestamp, value at index 1 is the comment text.

Simutool.CSVprocessor

`Parser.java`

```
public FileDTO parseFilesForPanels(String type, Reader r)
 Takes as argument the type of file (sensor, simulation, curing cycle, comments) and
a Reader with this file.

Uses univocity csv parser.

Output of parsing is a List<String[]> rows.

We take away everything but first three indexes of each line (to sift out unneeded
spaces or empty values on the end of line). Also take away first line that stands for
column names.
```

```
public void parseMetadata(Simulation s, FileReader path)
```
If path is null, takes file default.csv in metadata folder.

First line is csv column names, the second line are values.

Order:

operators,oven,material,tool,name,start_time,end_time,timezone,description,id

`ExperimentSaver.java`

```
public long normalizeTimeStamp(String inputTime)

Convert a string with a datetime to a long (timestamp).
```

```
public void savePanels(Simulation s)
Takes all data from influx.
Writes metadata to a csv (for internal use) and json (for uploading).
Create FileDTO instances from collected data and write it to file with writeCSV().

public void writeCSV(FileDTO file, String simulationName)
Remove all non-literal and non-digit characters from file name.
If directory with experiment name doesn't exist, create it, otherwise overwrite.
```

Application.properties

server.port = 8090
Port Spring Boot app is running on

grafana.host = localhost:8080

Port Grafana is running on
For changing you need to change http_port variable in src/github.com/grafana/grafana/conf/custom.ini

influx.host = http://localhost:8086
Port of influx database
For changing you need to uncomment and change bind-address variable in influxdb/influxdb.conf

influx.user = admin influx.password = 12345
Influx credentials

BUILD_DIR = ../
Relative path from java app root to project root directory.
In development mode, java app root is java-app folder, thus, value is set to "../" by default. In production java root is your .exe file.

saveCSVfolder = ${BUILD_DIR}output
directory for storing saved experiments

metadataFolder = ${BUILD_DIR}configs
Directory for default metadata files

influxStarter = ${BUILD_DIR}influxdb/
Folder with .exe file that launches influx server.

grafanaStarter = ${BUILD_DIR}src/github.com/grafana/grafana/bin/ Folder with .exe file that launches grafana server.