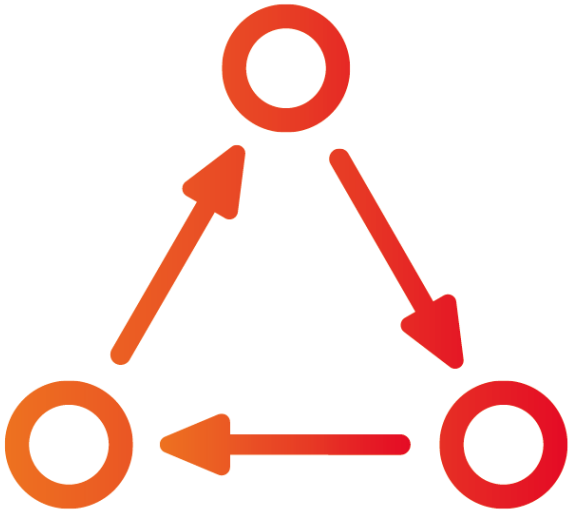
Tactalyse-2

Design Document



Sangrok Lee (S3279480)

Bianca Raețchi (S4755138)

Matteo Gennaro Giordano (S5494974)

Mikko Brandon (S3781356)

2023

RUG

Contents

Glossary

Term	Definition
(The) service	An abstraction of the PDF Generator application.
Graphs/Plots	Radio charts and line plots containing player statistics.
Coaches	Football coaches of professional football clubs.
Players	The football players the coaches request Tactalyse's service for.
(The) client	Tactalyse's owner (or Tactalyse's clients if specified)

Introduction

Tactalyse is an organization that aims to provide data reports about football players. The data report contains statistics and graphs based on the football players' match data. To build those graphs and statistics, coaches who are customers of Tactalyse upload the players' data in excel file format on the company's website. Then, employees of Tactalyse receive the data and upload it to a separate page on the website, accessed only by them. The website generates a data report. This generated data report will be e-mailed to coaches for improving the players' performance. In this project, our main task is to receive data from the frontend service, create related graphs and plots based on the received data, and create data reports from these graphs.

The company already has a completed mechanism that receives data and creates related graphs and reports. However, the original system has several bugs, and is written with bad code logic and with no architecture in mind. Additionally, the used libraries are not optimized, and very slow. Since data report generation speed has a great impact on the company's work performance, the owner wants us to build their project from scratch, rather than extend the original project.

In this document, the architecture and design of our project will be described.

Stakeholders

We discussed our stakeholders separately in our requirements document. However, in order to motivate our choices for architecture and design, we felt it would be useful to reiterate them in this section.

- **Tactalyse employees**: They will be the main users of our app and they will send the clients the generated reports. Therefore, creating an *intuitive* and *efficient* app for the employees to work with is going to reduce the potential for errors in the future and should be kept at the forefront of our architectural design.
- **Owner of Tactalyse**: this stakeholder owns the app. In order to keep it satisfied, we should *stick to the deadlines* and the *app features* they require of us. The owner particularly values their time, so the app needs to be *fast* as well.
- **Coaches**: they will receive the reports generated by the employees. Since they are the main clients of Tactalyse, it is crucial to make sure this stakeholder's requirements are at the top of our priority list. We can expect that they will favor data reports with a *clean layout*, so the information displayed will be easy and straightforward to understand.
- **Players**: the players are indirect stakeholders of our application. They will be impacted most by their coaches' interpretation of our data reports. In order to have a better understanding of their current athletic abilities, providing a *comprehensive selection* of such reports would be the most beneficial component.

Architectural Requirements

This section describes the architectural requirements the client imposed on us. These will not be functional requirements as described in the requirements documents, but rather requirements that significantly affect the architectural design of the project.

- **Python:** The client wants us to write a backend for the app in Python. The existing code was written as such, so there is some familiarity with the language within Tactalyse. Since the code is going to be maintained by the company after we deliver it to them, it is important that the coding staff understands how it works. Writing the code in Python facilitates this.
- **Flask:** With similar reasoning, the client proposed Flask as our backend framework.
- **Excel data:** To create the PDF reports, we need to process football data into a graphic format. The data comes from a service named Wyscout, which provides it in Excel sheets. Due to financial constraints, it is not possible for the client to gain access to an API that would obtain these files directly from Wyscout. As such, our design needs to allow for the input of Excel sheets, which will then be processed further.

Technology Stack

This section describes the technologies we used to best fit the client's needs, and our motivation for their selection.

Language

The project description listed Python as a mandatory language for the project. After discussing the matter with the client, they informed us that we are free to use any technologies we see fit, as long as we can motivate our choice. However, they did highlight that there is existing familiarity with Python among Tactalyse's employees. As such, we decided to stick to Python, but to motivate the choice further.
One clear advantage of the language is that it's relatively intuitive. Since it is a high level programming language, most of its written code resembles natural language. Its high-level aspect makes it easy for anyone with programming experience to maintain code written in the language. As such, regardless of existing experience within Tactalyse, the code will be easy to maintain in the future, which fits our task of creating more **maintainable** code with **better structure**.
Another feature unique to Python is its enforcement of code formatting. Instead of using brackets and semicolons, Python uses spacing and colons to create separation and cohesion between lines of code. If code is poorly formatted, it will simply stop execution during runtime. This adds to the previously stated benefit to allow for greater **readability** and **maintainability**.
Finally, Python has extensive library support, among which are many **data-oriented** libraries. This naturally includes libraries for plotting, graphing, data manipulation and writing to specific

file formats. Since the project requires the output of data in multiple forms, such as graphs and dataframes, it is reasonable for us to use this language for our project.

Framework

The client suggested we work with Flask for our backend framework, for reasons similar to the ones they mentioned for the use of Python. Once again, we chose to follow their suggestion, as well as to motivate it further. When it comes to Python frameworks, the most popular choices are Flask and Django. To illustrate our decision-making process, we will discuss the similarities and differences between the two frameworks.

Firstly, for reasons mentioned in our choice of programming language, we believe it is important that the framework we use is **intuitive** and **easy to understand**. It allows for greater maintainability by employees of Tactalyse in the future. Flask is a microframework, meaning it has minimal built-in functionality. Django is a full-stack web framework. Its philosophy is "batteries included"; it has as much functionality as deemed required for most web applications. Although this allows for more variety "out the box", it makes it harder to comprehend all functionality. As such, it is easier for someone unfamiliar with the framework to wrap their head around Flask than it would be for Django.

As mentioned, Django has more built-in functionality. However, for the purposes of our project, a lot of the built-in functionality is not required. Furthermore, if the client wishes to develop the code further to include more technologies and functionality in the future, it is not the case that it would be impossible to implement in Flask. In fact, due to Django's rigid nature, it might be easier in some cases to maintain new technologies in Flask. Therefore, we believe Flask to be the right choice regarding its fit to the currently planned **required functionality**.

Finally, the company's employees have some **existing experience** using Flask. If we were to use Django for our project, it would require more effort on Tactalyse's part. We want to keep future effort and development time to a minimum. With this in mind, Flask would be the logical choice.

All in all, we believe Flask to be the right choice for our project's web framework.

Data

Our software takes data from two sources: a data file containing information about a football league's players, and a data file containing match information about a specific player in that league. Both of these files are uploaded in .xlsx format, i.e. Excel worksheets. As mentioned in the architectural requirements, the files originate from a service named Wyscout. Clients of Tactalyse share these files with Tactalyse when requesting a PDF, after which an employee sends the files to our backend. The input data format was not negotiable. As such, we stuck with Excel worksheets.

The data files from Wyscout are constantly updated. This makes it so that it is unlikely for two uploaded data files to be the same in terms of the data they contain. As such, the use of a database for data preprocessing was deemed unnecessary. In discussion with the client, we found that if a database were to be used, it would be to provide read-only access to previously

uploaded data for our client. However, due to the fact that it would not directly contribute to the PDF generation process, we decided to leave the potential use of a database for the future.

Libraries

In this section, we describe the reasons why we chose the following libraries for our project. Some libraries are used by the original project, but there are also new libraries.

- **Pandas**

A big reason we chose Pandas was that the existing code used Pandas to process raw data. The client wanted continuous code maintenance even after we completed the project. Therefore, we chose Pandas, which the client is already familiar with. Moreover, Pandas provides faster data processing speed than Excel. Since one of the most important requirements of clients is to speed up the time to create data reports, data processing speed is a very important part.

- **Seaborn**

Seaborn is a Python data visualization library based on matplotlib. Seaborn allows a concise but limited approach to quickly visualizing datasets with nicer-looking style defaults than Matplotlib. Therefore, it is easy to write concise code and generate more plots in less time. Since we need to work on increasing the efficiency of our code and reducing the time it takes to produce results, we adopted seaborn as our graph generation library.

- **Matplotlib**

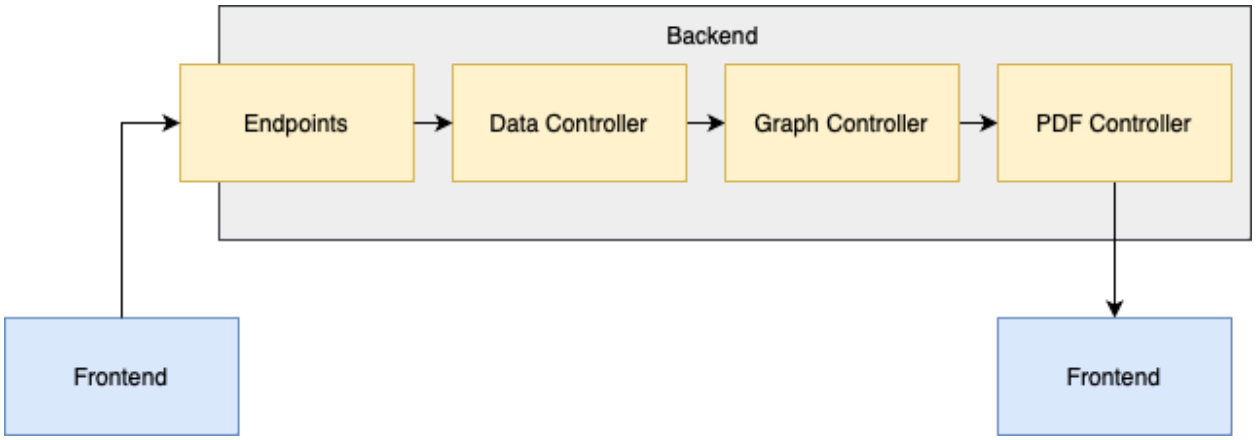
Seaborn has no standard option for radio charts, which we need to include in our document per the requirements. As such, we will use matplotlib to create these plots.

- **Fpdf2**

For reasons similar to Pandas, since the existing code generates data reports with the pyFPDF library, we decided to use this library for PDF generation. In the beginning, we agonized over the choice between pyFPDF and PDFlib. Since the generation speed of the document is slower with PDFlib, we chose pyFPDF. The library has all the features we need in our project including image support, which is the most important part for the data reports.
The original pyFPDF release that is out as of the development of our application is quite slow. Generating a two page PDF with just two PNGs included took roughly 27 seconds. Since the client's main requirement was increased speed of the application, we decided to use fpdf2, a fork of the original pyFPDF release. It uses a byte array buffer, which speeds up the output of a pdf file significantly. Specifics on why this is faster can be found here.

Architectural Overview

This section describes the architectural decisions we made to guide the coding process, and the overall structure of the application. For better comprehension, we present the following conceptual diagram of the data flow:



Design Patterns

REST API

We decided to create an API to allow access to our services from any other component in the greater system. It makes our services more flexible than if we were to deliver them as Python scripts that would be plugged into the system wherever necessary. As such, we adhered to REST API design, facilitated by Flask. This entails the use of HTTP verbs and response codes.

MVC

To guide the coding process and ensure best practices, we decided to use the Model-View-Controller (MVC) pattern. We deemed it appropriate considering the functionality and structure of our software system.

- **Model**

Per definition, the Model contains all core functionality and data access code. In our software, this entails data extraction/manipulation, graph generation, and PDF generation.

- **Controller**

The controller serves as an interface between the model and the view. The view signals the controller that something has changed. The controller makes the appropriate call to the model's functions according to the information received from the view. The model processes the passed information, and returns it to the controller. The controller currently only includes one API endpoint. More on this can be found in the Module View and Execution View sections.

- **View**

The view is the user interface of the application. It is updated through the controller. In our application, this is currently done with simple messages signaling success or failure to the user. The implementation of the view is part of Tactalyse-1's project. As such, we will not go into the specifics in our document.

Object Oriented Design Patterns

We employed the **abstract factory** design pattern in the graph generator module. We believed it to be a good fit due to the fact that we plan on allowing for many different kinds of plots to be defined. To have a good overview of which plots can be instantiated, and to keep the specifics of the instantiation of plot objects under the hood, we believed having a common interface for each plot super-class would be beneficial.

API Specification

This section details the functionality of our PDF generator API, and its endpoints.

Maturity level: The API functions at maturity level 3. It makes use of HTTP (level 0), it uses uniform resource identifiers (level 1), it uses HTTP verbs (level 2), and it uses content negotiation in the form of allowed content types (level 3).

Server: http://localhost:5000/pdf

Parameters:

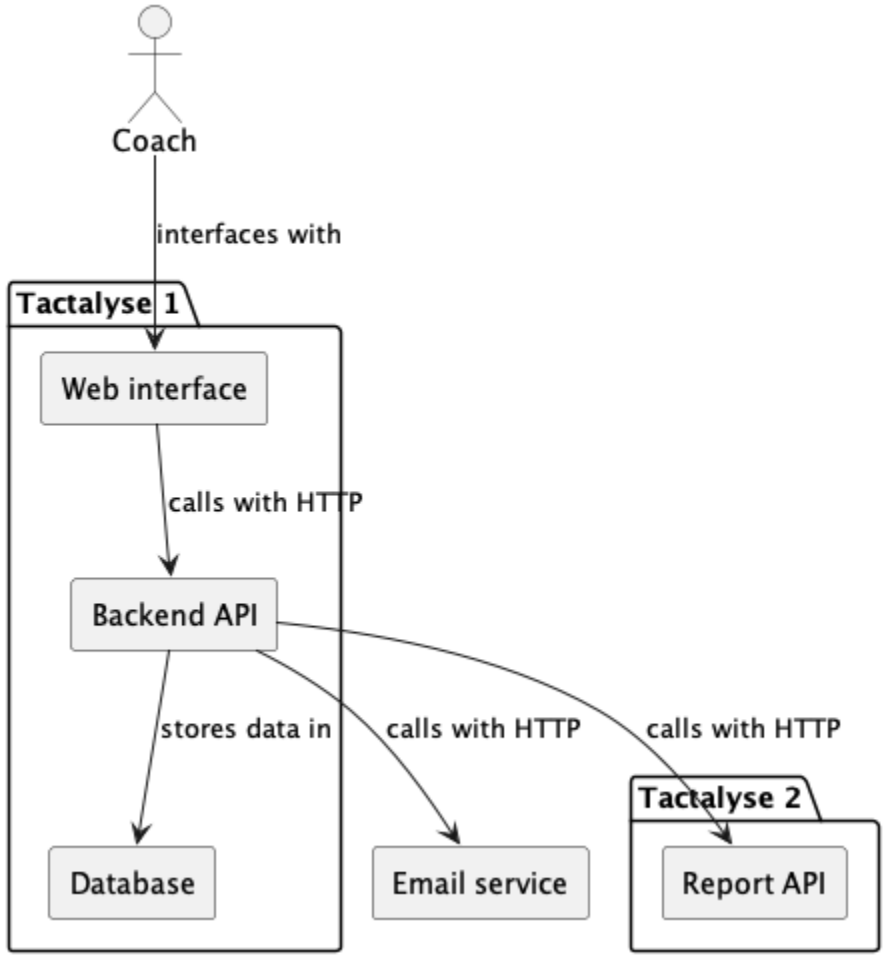
- **league-file**
  - .xlsx format data file containing data about players in the league the file represents
  - In query
- **player-file**
  - .xlsx format data file containing match data of the player the file represents
  - In query
- **player-name**
  - String representing the name of the player whose match data is contained in **player-file**
  - In query
- **start-date**
  - String (date?) representing the starting date of Tactalyse's services for the specified player
  - In query
- **end-date**
  - String (date?) representing the ending date of Tactalyse's services for the specified player
  - In query

Endpoints:
Here, each endpoint is listed with its HTTP methods. Required parameters are written in bold.
/pdf

- POST
    - Retrieves a PDF based on passed parameters
    - Parameters: **league-file, player-file**, **player-name**, start-date, end-date
    - Responses:
        - ➢ 200
            - Case: PDF report successfully created.
            - Body: PDF report in byte format.
        - ➢ 400
            - Case: Bad request (files did not match what was expected)
        - ➢ 5XX
            - Case: Server error

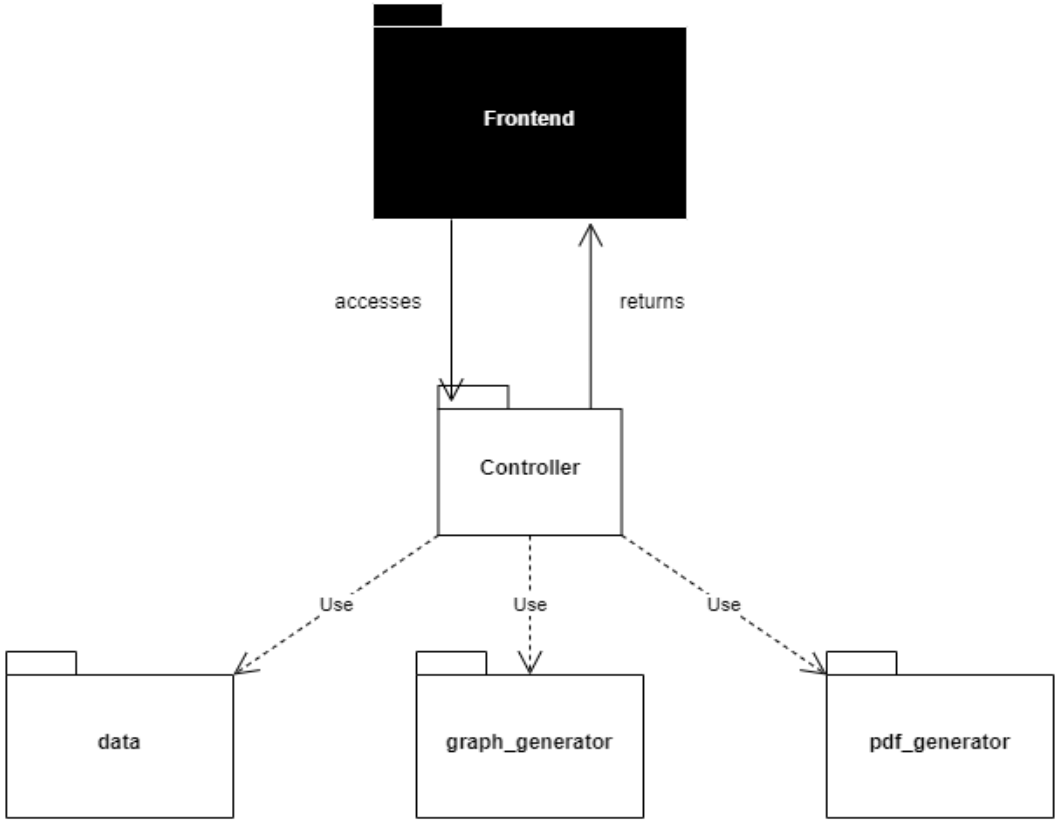
Conceptual View

This section gives a conceptual overview of the whole system.
The system as a whole serves as a website for Tactalyse as a company. The development of this website has been divided over two teams: Tactalyse-1, and us, Tactalyse-2. Tactalyse-1 is responsible for the website's entire frontend and additional backend functionality. We are responsible only for the backend of the PDF report generator. We have decided to create an API, which is called from the backend developed by Tactalyse-1. As such, it can be seen as an extension of Tactalyse-1's backend. The component diagram shows a high-level design of the system, with each team's components divided through a border with the team's name. Arrows roughly indicate dependencies between components.



Module View

This section gives an overview of the module design of our project.
To adhere to the MVC pattern, we have decoupled the API endpoint in the controller from the core functionality of the application. This is visible through the separation of the `controller` module, which only contains the endpoint, and `service` classes that make calls to functions from other modules.
The model is represented through three modules: `data`, `pdf_generator` and `graph_generator`. `data` contains functionality related to extracting and processing data from Excel files. `graph_generator` contains functionality related to converting the processed data into relevant graphs. `pdf_generator` contains functionality related to generating a PDF report from input. `graph_generator` will need data in a dataframe structure in order to generate the graphs. This data will be provided by `controller` after passing the received excel files through `data` function calls. In turn, `pdf_generator` will need graphs in an image format in order to generate the reports. These will be provided by `controller` after generating them through `graph_generator` function calls. As such, `data`, `pdf_generator` and `graph_generator` are fully decoupled. `controller` is the only module with dependencies on each of the other modules. The frontend queries the API, and gains access to its functionality that way. It should receive a generated PDF through a single call to the API.

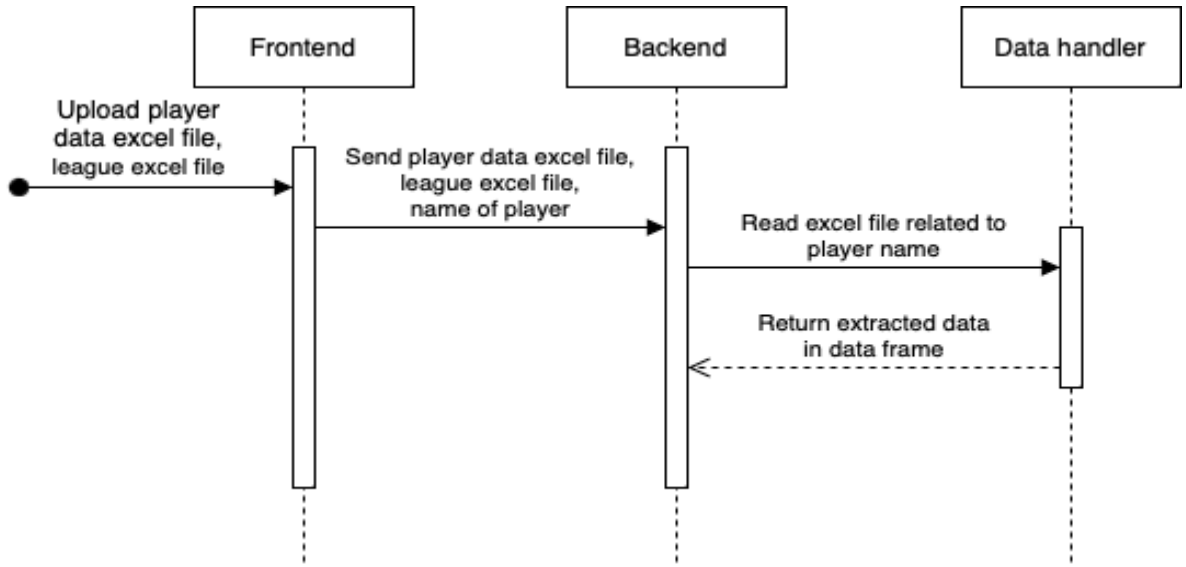


Execution View

In this section, we describe the sequential process of our project. As explained in the introduction, coaches upload data of football players in Excel file format, and employees of Tactalyse receive these data files. These files are passed through the frontend service to the backend. We have three modules: data, graph generator, and PDF generator. The following diagrams show what a sequence in our program looks like at each step between modules.

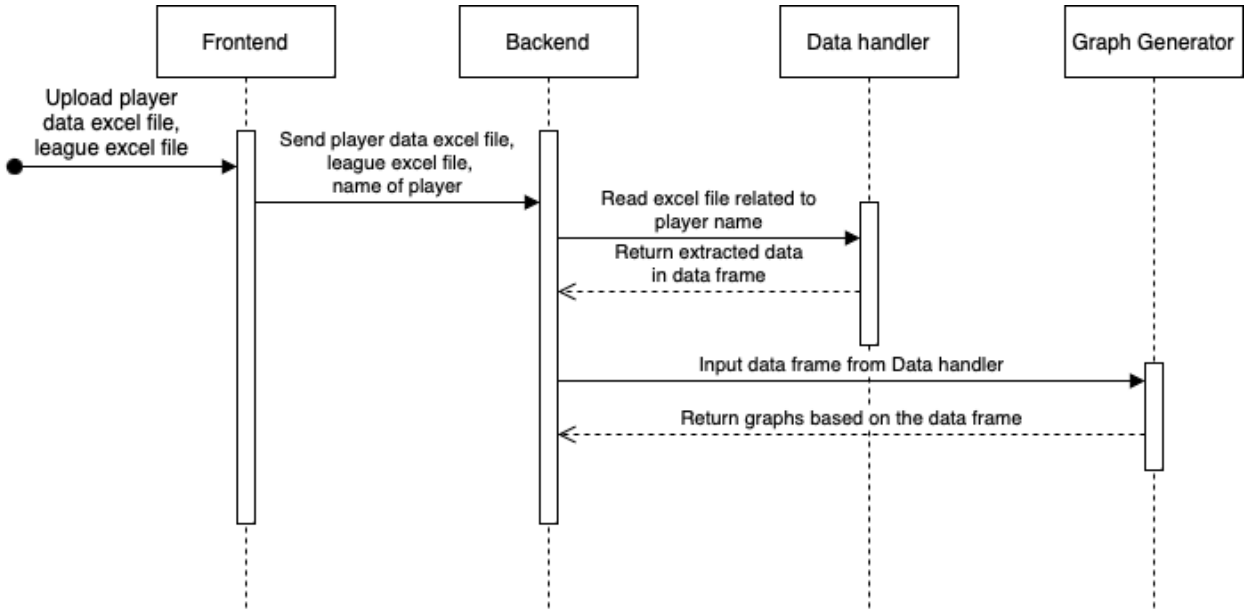
- ## Backend ↔ Data

This diagram details the first phase of the execution of our program: data processing. Raw data is received from the frontend, which is then forwarded to the data module in order to extract the data into dataframes.



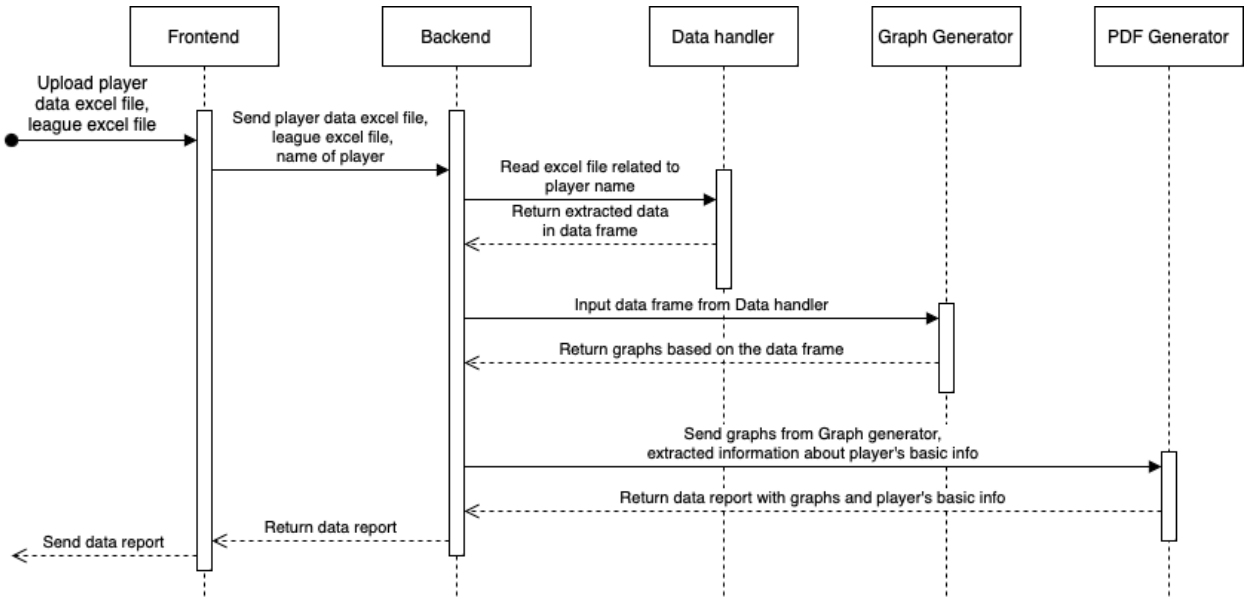
- Backend ↔ Graph Generator

After the data has been processed, it can be sent to the graph generator in order to create graphs. These graphs are then returned to the controller for further use.



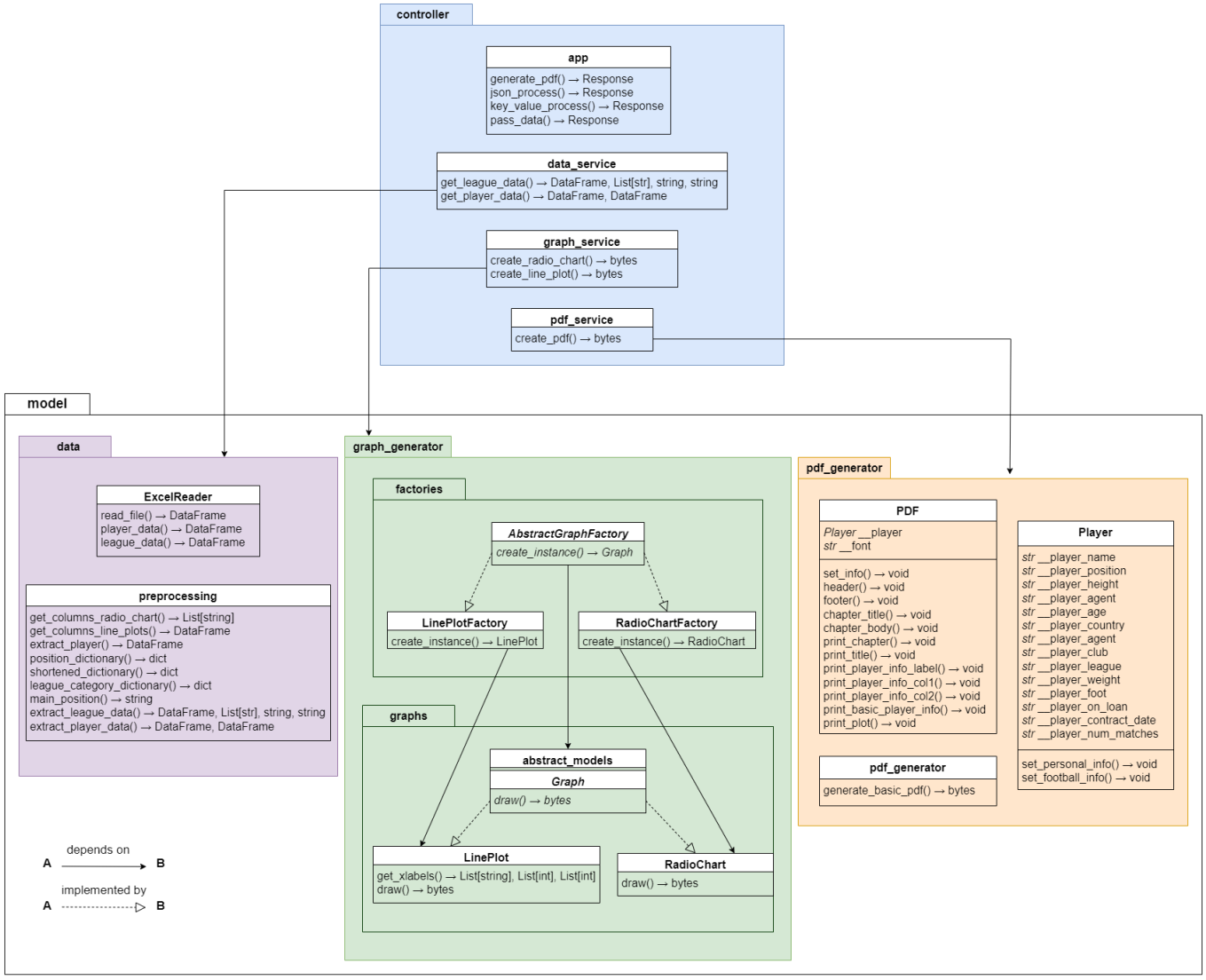
- Backend ↔ PDF Generator

After graphs have been generated, they can be passed to the pdf generator along with the processed data, so that they can be displayed in a PDF document. This document is returned to the backend controller after generation. Finally, the PDF document is sent to the frontend, which handles it further.



Code Allocation View

In this section, we provide an overview of the code base in a class diagram. Adhering to the structure defined in the higher-level views, we wrote functions in a manner that would allow for minimum coupling between modules. Each function is defined as follows: `name() -> return_type`. Python allows for the use of access modifiers in classes through underscores: `__temp` indicates a private variable, `_temp` a protected variable, `temp` a public variable. Field variables are defined as *`type`* `(__)name`. Abstract classes and functions are indicated by the name being italicized. For classes (indicated with capital letters), it may be assumed that all private field variables have getters.
In case better readability is desired, we have sectioned the same diagram without dependency arrows, and placed it in Appendix B.



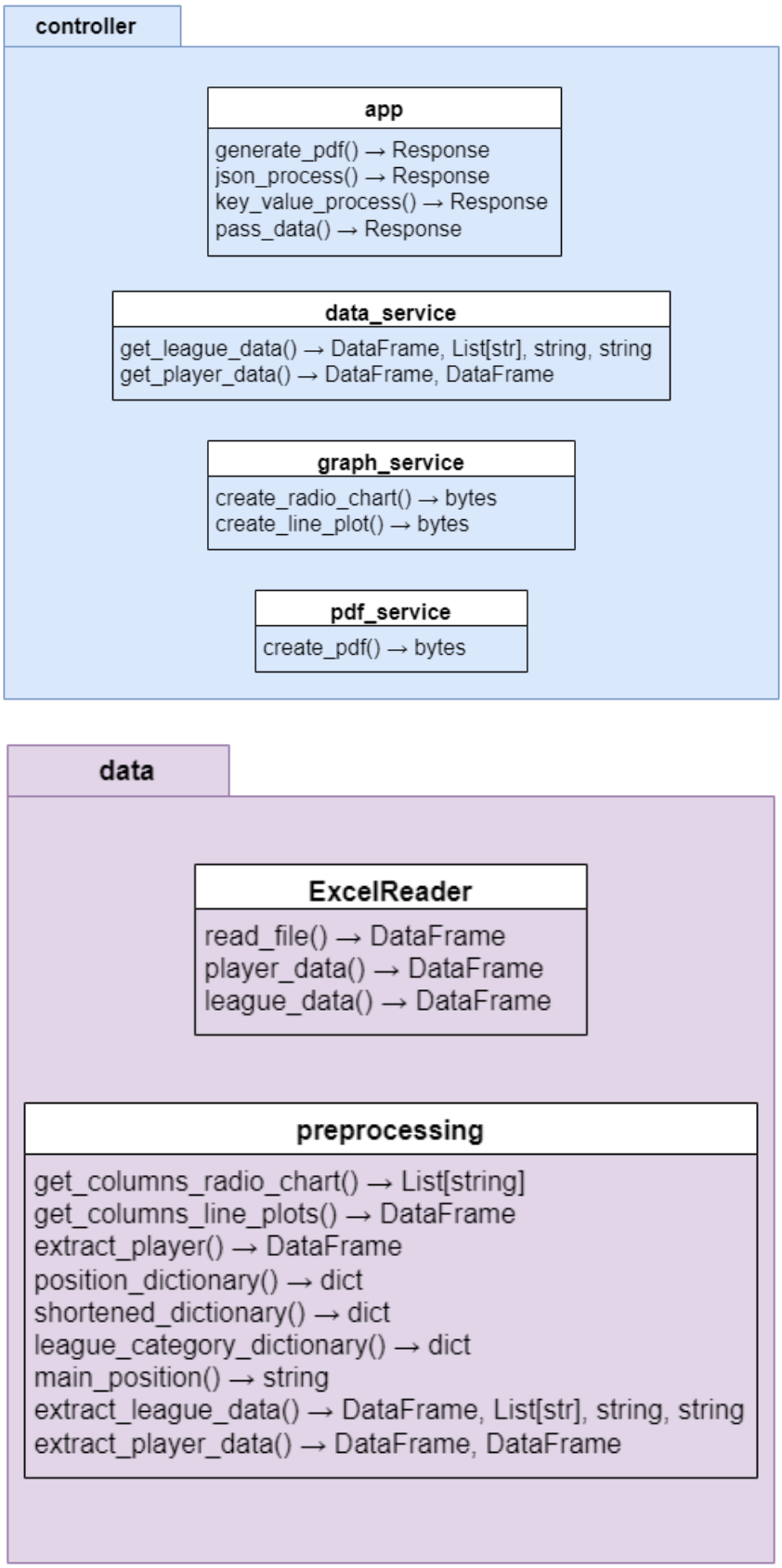
Appendix

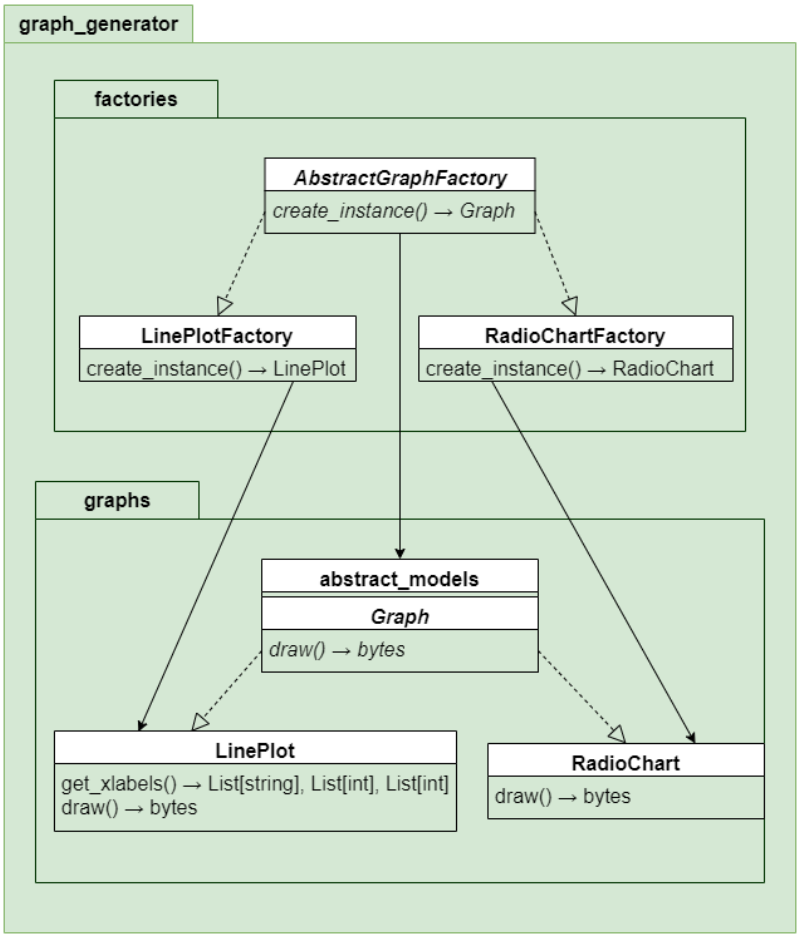
A. Change Log

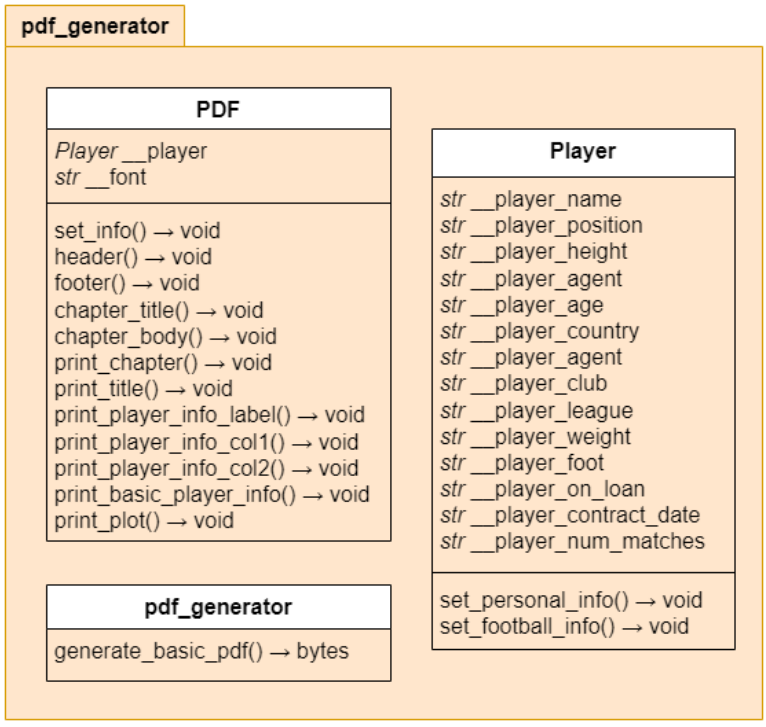
- 17.March.2023 **Sangrok**, **Mikko**, **Bianca**: Add sketch of the architecture.
- 20.March.2023 **Sangrok**: Add sequence diagram for Execution View
- 20.March.2023 **Mikko, Bianca**: Add UML diagram for Module View
- 21.March.2023 **Sangrok**: Add Introduction section of the document.
- 23.March.2023 **Mikko**: Stakeholders (pasted), significant requirements, technology stack (language, framework)
- 23.March.2023: **Sangrok**: Add Libraries section(Pandas, pyFPDF)
- 24.March.2023: **Sangrok**: Add contents in pyFPDF, Add Seaborn section, Execution View.
- 24.March.2023: **Mikko**: Data, Design Patterns, API Specification, text for Conceptual View and Module View.
- 28.March.2023: **Sangrok**: Modify and add some contents in Libraries and Execution View.
- 31.March.2023: **Sangrok**: Add table of contents and Glossary, Fix Introduction, Technology Requirements, Technology Stack based on the feedback.
- 01.April.2023: **Mikko**: Modify conceptual and module diagrams, Code Allocation View text.
- 01.April.2023: **Sangrok**: Add high-level diagram of our project, Fix Execution View by breaking down the original diagram.
- 02.April.2023: **Sangrok**: Fixed typos in the diagram(Backend ↔ PDF generator).
- 02.April.2023: **Mikko**: Went over Sangrok's changes.
- 03.April.2023: **Mikko**: Added class diagram, final changes to document.

B. Class Diagram

The text in the class diagram ended up being too small to read when not zoomed in. However, we did not find it relevant to add the same diagram divided into sections, as it adds nothing in terms of information. As such, we decided to showcase the diagram in sections in the appendix, without dependency arrows.


controller
app
generate_pdf() → Response
json_process() → Response
key_value_process() → Response
pass_data() → Response
data_service
get_league_data() → DataFrame, List[str], string, string
get_player_data() → DataFrame, DataFrame
graph_service
create_radio_chart() → bytes
create_line_plot() → bytes
pdf_service
create_pdf() → bytes
data
ExcelReader
read_file() → DataFrame
player_data() → DataFrame
league_data() → DataFrame
preprocessing
get_columns_radio_chart() → List[string]
get_columns_line_plots() → DataFrame
extract_player() → DataFrame
position_dictionary() → dict
shortened_dictionary() → dict
league_category_dictionary() → dict
main_position() → string
extract_league_data() → DataFrame, List[str], string, string
extract_player_data() → DataFrame, DataFrame


graph_generator
factories
AbstractGraphFactory
create_instance() → Graph
LinePlotFactory
create_instance() → LinePlot
RadioChartFactory
create_instance() → RadioChart
graphs
abstract_models
Graph
draw() → bytes
LinePlot
get_xlabels() → List[string], List[int], List[int]
draw() → bytes
RadioChart
draw() → bytes


pdf_generator
PDF
Player __player
str __font
set_info() → void
header() → void
footer() → void
chapter_title() → void
chapter_body() → void
print_chapter() → void
print_title() → void
print_player_info_label() → void
print_player_info_col1() → void
print_player_info_col2() → void
print_basic_player_info() → void
print_plot() → void
Player
str __player_name
str __player_position
str __player_height
str __player_agent
str __player_age
str __player_country
str __player_agent
str __player_club
str __player_league
str __player_weight
str __player_foot
str __player_on_loan
str __player_contract_date
str __player_num_matches
set_personal_info() → void
set_football_info() → void
pdf_generator
generate_basic_pdf() → bytes