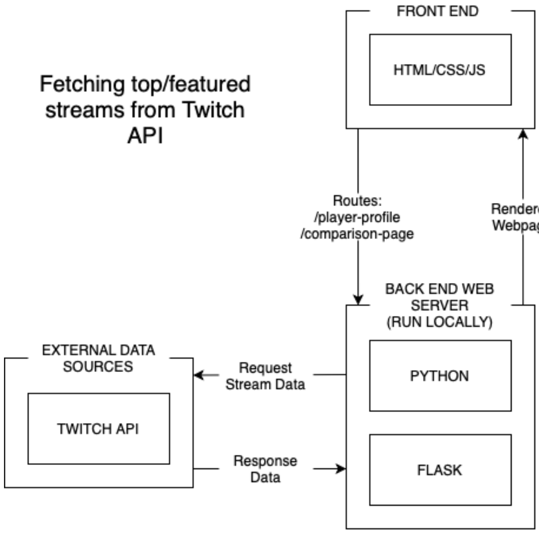
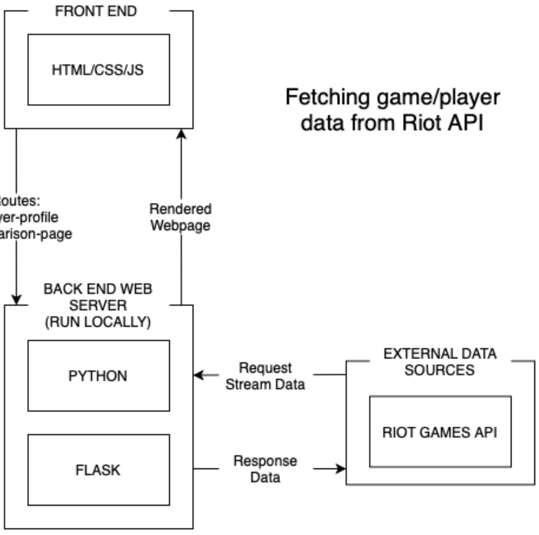
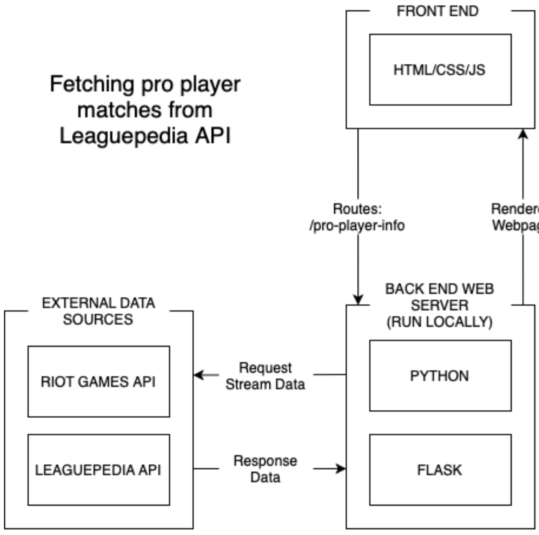
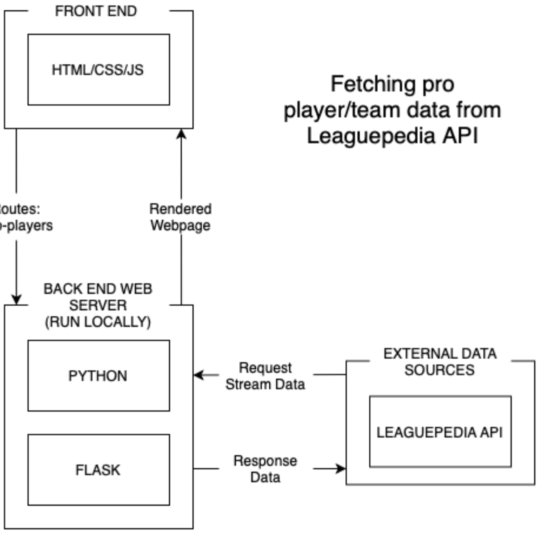
Girra Gang SENG2021 Deliverable 2

Part 1: Software Architecture


Fetching top/featured streams from Twitch API
FRONT END
HTML/CSS/JS
Routes: /player-profile /comparison-page
Rendered Webpage
BACK END WEB SERVER (RUN LOCALLY)
PYTHON
FLASK
Request Stream Data
Response Data
EXTERNAL DATA SOURCES
TWITCH API







External Data Sources

The external data sources that will be leveraged include the Riot Games API, the Twitch API, and the Leaguepedia API. The Riot Games API is used to access player data such as match histories as well as game data including assets, the Twitch API provides data on live channels to feature on the homepage, and the Leaguepedia API provides information on the professional players.

Chosen Technologies and Frameworks

With regards to the front-end framework, vanilla HTML and CSS will be used to create the static layouts of each page, and occasionally external JavaScript libraries such as Chart.js in conjunction with jQuery will be used in creating more dynamic elements on webpages.

Python and Flask will be used to obtain data from external data sources through REST API calls. The frontend will query the Flask backend, which will call a Python function to obtain the pertinent data from a third party and use the results to alter a Flask template which will be returned to the user's browser to be displayed.

Application and Justification of Chosen Technologies

Vanilla HTML/CSS vs Bootstrap

	Simplicity	**Creative Control**	**Time**	**Documentation**	**Size**
HTML/CSS	**Easy to pick up.**	**Full creative control but may require 'hacks' to achieve certain results.**	**Certain layouts can require lots of work.**	**Very well documented.**	**Often small.**
Bootstrap	**Requires background knowledge of HTML/CSS.**	**Slightly restrictive and often results in a similar aesthetic to other Bootstrap sites.**	**The same layouts can be achieved very quickly.**	**Very well documented.**	**Can have excessive bloat if features go unused.**

Objectively, both HTML/CSS and Bootstrap are legitimate choices of front-end design tools, and both offer different advantages. For example, whilst Bootstrap is designed to be responsive by default, responsive CSS can take a lot of effort to design effectively. On the other hand, CSS can be used to create much more unique design.

Ultimately, for the purposes of this project, the web application's front-end will be designed using vanilla HTML/CSS due to its simplicity specifically its lack of need for prerequisite knowledge, as well as prior experience. The CSS will be designed with a desktop-first philosophy and should function well on popular desktop-class browsers. The front-end will also employ an external JavaScript library, Charts.js, alongside jQuery to provide visual representations of certain data sourced from external data sources.

Python vs JavaScript for Web Framework

	Learning Curve	Error Handling	Stack Relevance	HTML Support	Speed
Python	Easy to pick up.	Excellent error handling.	Half stack, backend is separate.	Harder to implement with HTML.	Slower than JavaScript.
JavaScript	Notably more difficult to pick up.	Good error handling.	Full stack JS, backend is the same as frontend.	Easy to implement with HTML.	Faster than Python.

Though JavaScript, specifically Node.js, is a popular language for the web, Python is a much simpler language with better tools for testing, error handling and debugging. JavaScript is very popular and has an extremely active community. It is also integrated into web browsers by default. It is important to note that both have an overlap in terms of project applicability, and both also have their own advantages.

A significant advantage for Python in regard to this project is the prior experience held by the development team, and as such, makes it a more suitable choice as the programming language for the web framework.

Flask vs Django

	Simplicity	Performance	Project Suitability	Default inclusions	Feature extensibility
Flask	Simple extension of Python.	Very performant.	Suitable for smaller projects.	Includes many fewer features out of the box.	Limited feature set.
Django	Steeper learning curve.	Performs well.	Suitable for large projects.	Includes many useful features out of the box.	Very customisable feature set.

Flask and Django are both very popular options for Python based web frameworks. Though this project does not benefit from it specifically, it is important to note that Django provides deep integration with a RBDMS, which Flask does not offer. Both of the above frameworks are valid choices for web frameworks. Ultimately, this project will be using a Flask back-end, due to its simplicity and prior experience among the development team.

Flask Templating vs jQuery vs React

	Simplicity	Dynamic Content	Time	Documentation	Size
Flask Templating	Requires pre-requisite knowledge of Flask.	Implemented by writing HTML as Jinja 2 templates.	Quick if using Flask for backend.	Well documented.	Negligible if using Flask for backend.
jQuery	Smaller learning curve.	Implemented by direct manipulation of DOM, can be slow.	Quick setup.	Very well documented.	Large.
React	Larger learning curve.	Abstracts manipulation of DOM, simpler.	Requires timely initial setup.	Very well documented.	Large.

In objective terms, Flask templating should only be used if the web framework is also Flask, and as such, it is a foolish option for dynamic content otherwise. jQuery and React, though they have their own strengths, can both be used for displaying dynamic content, such as large match histories on a player's profile. However, it should be noted that due to the fact that jQuery directly modifies the DOM, it can result in clunky and slow behaviour, whereas React obfuscates this manipulation by rendering its elements behind the scenes.

Since this project is using a Flask backend, the dynamic content will be created by altering the HTML files to use Jinja 2 templating for use with Flask. This will make it so an external framework like React is wholly unnecessary and the large size of jQuery will only be used when absolutely necessary, rather than on every single page.

Platform Choice

Local Deployment vs Cloud Deployment

	Cost	Complexity	Performance	Feature Support
Local Deployment	Cheaper.	Very simple.	Based on host machine.	Able to customise to fit project needs.
Cloud Deployment	Expensive. (Often cheaper for students)	Comparably complicated.	Varying, but usually faster than localhost.	Dependent on external support.

DigitalOcean vs AWS vs Google Cloud vs Azure

	Cost of Standard Plan	Global Coverage	Industry Usage
Digital Ocean	**$5/mo.**	**12 data centres globally, none in Australia.**	**Mainly used by smaller companies and freelancers.**
AWS	**$8/mo.**	**Total unknown, 3 availability regions in Sydney.**	**Used largely by enterprises, Netflix, Facebook, etc.**
Google Cloud	**$6/mo.**	**22 regions supported in 200+ countries. Many features available in Australia via Sydney.**	**Used by PayPal and HSBC, as well as other large companies.**
Azure	**$8/mo.**	**58 regions supported in 140 countries. 4 locations in Australia.**	**Used by Adobe, BMW and many other large companies.**

For the scope of this project, cloud deployment is excessive. However, in terms of industry relevance, cloud deployment services such as DigitalOcean, AWS, Google Cloud and Microsoft Azure are very relevant in industry and should be considered as alternatives. Ultimately, this project will be run on a local machine by running the Flask server on the same machine that will be accessing the site, simply because of its scope.

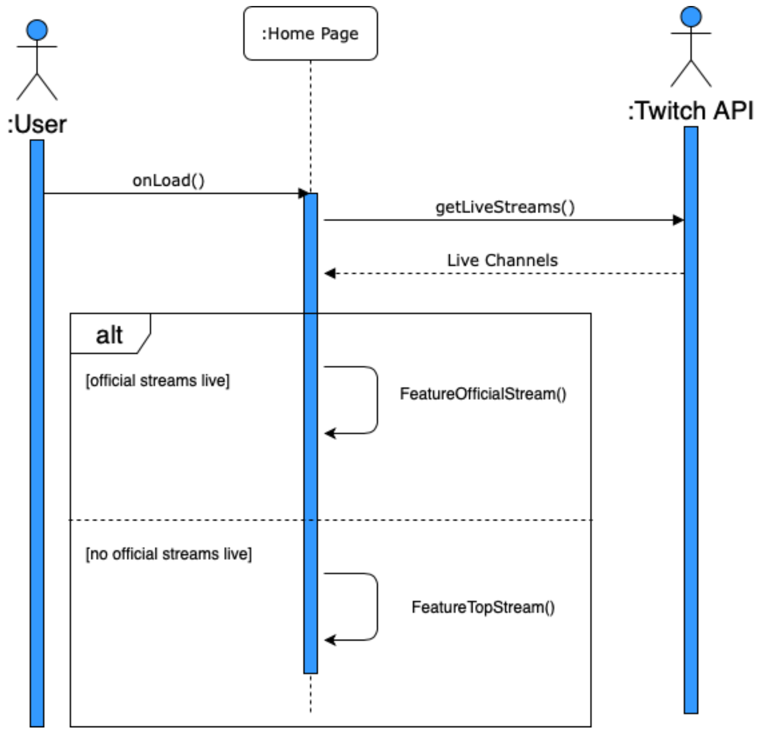
Summary

The software architecture for this project includes a frontend built in HTML (written as a Jinja 2 template) and vanilla CSS, a Python based backend, specifically Flask as a web framework, and Flask's in-built templating engine for the generation and rendering of dynamic content.

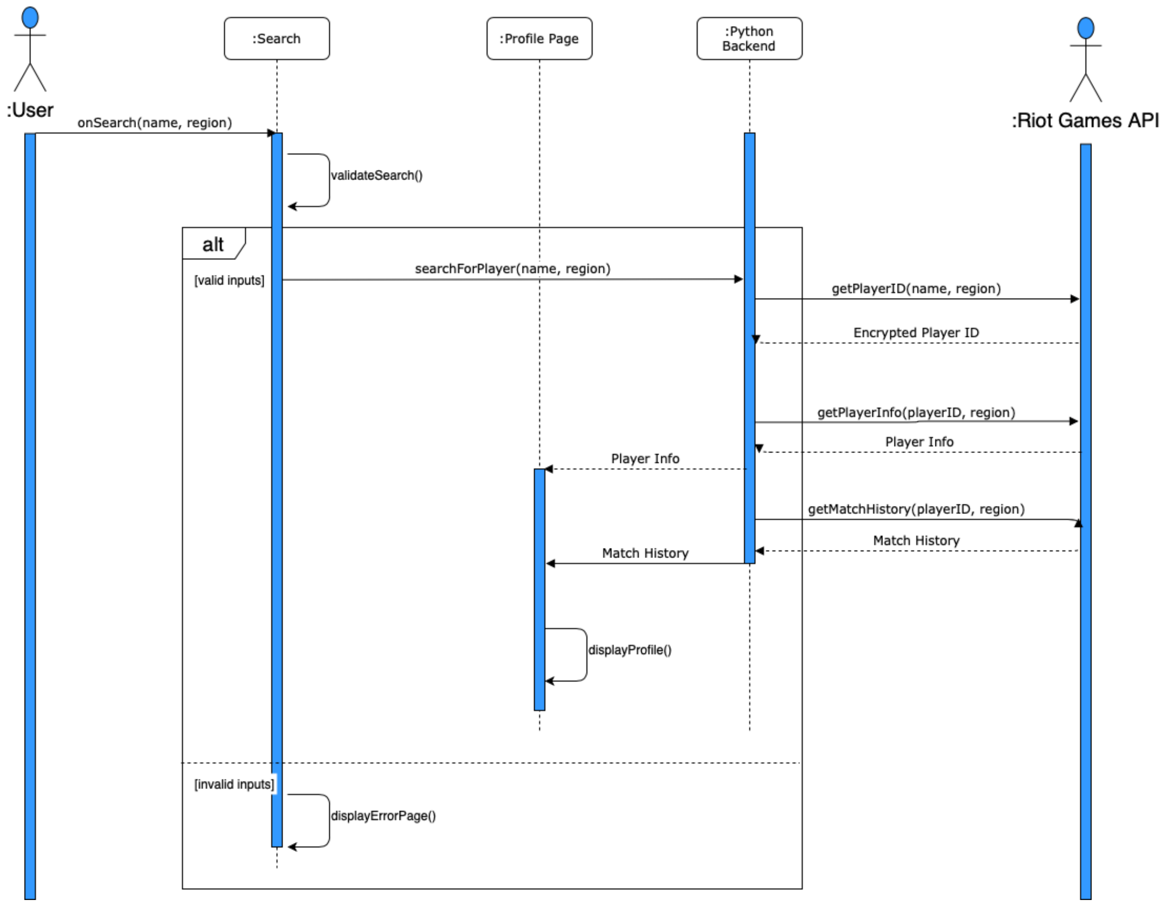
An example of how this software architecture interacts, is as follows. When a user requests to see a profession player's match history, the frontend will send a query to Flask with the professional player's identifier, where it will be passed to a Python function that will make a call to the Leaguepedia API to obtain their in-game name(s). If there are multiple names that are returned, each of these will be input into the Riot Games API, in order to get an encrypted account ID, which will be used in accessing the match histories of these accounts from the past six months. The Python backend will format and sort this data as required, then return it to Flask. This data will then be processed through the HTML/CSS template, and ultimately returned to the user.

Part 2: Initial Software Design

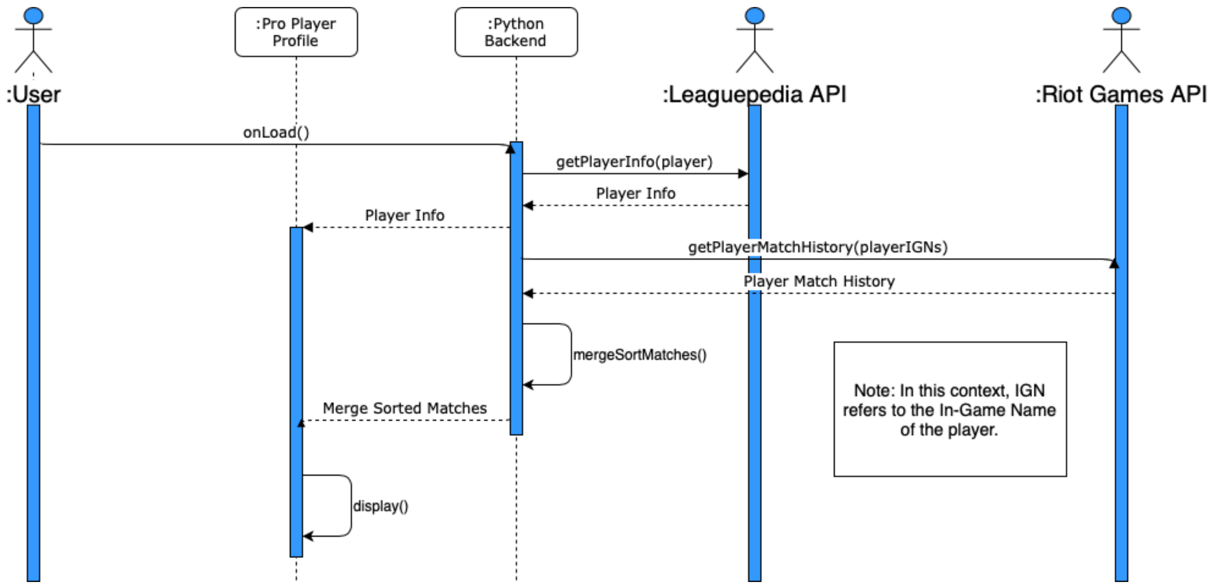
Display of live professional play on homepage.



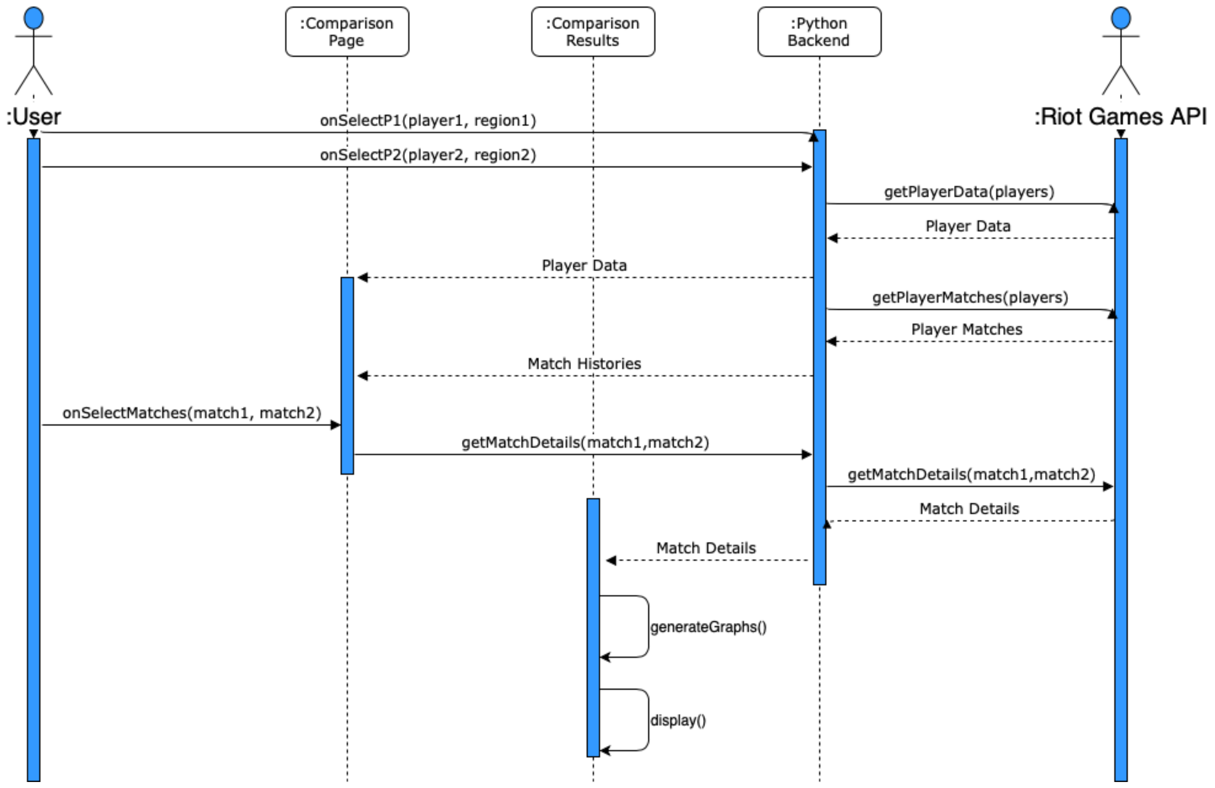
Searching for a specific player's match history.



Searching for a professional player’s match history.

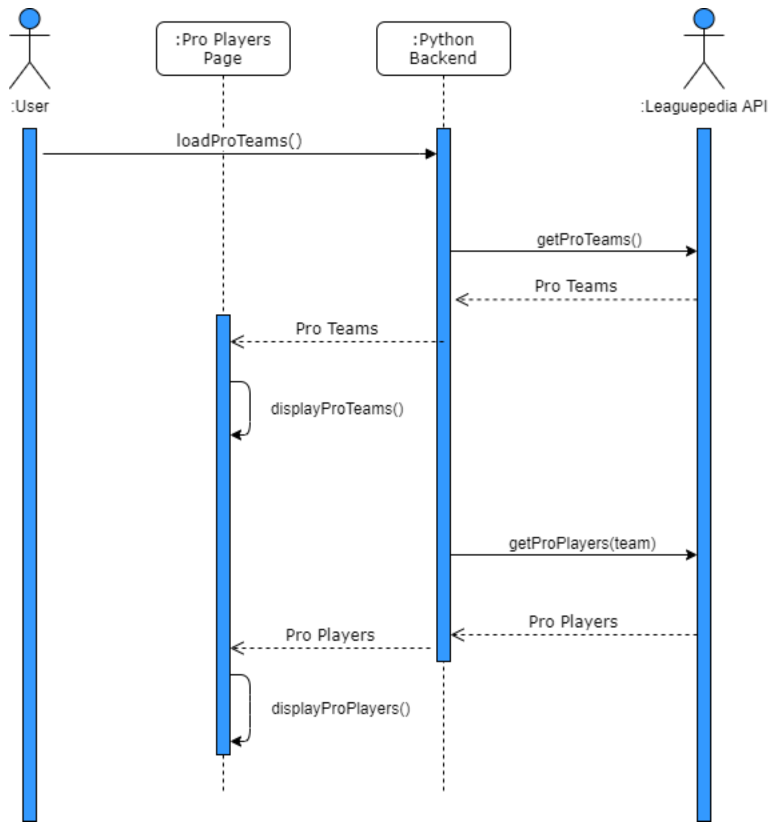


Compare two player’s specific matches.[1]

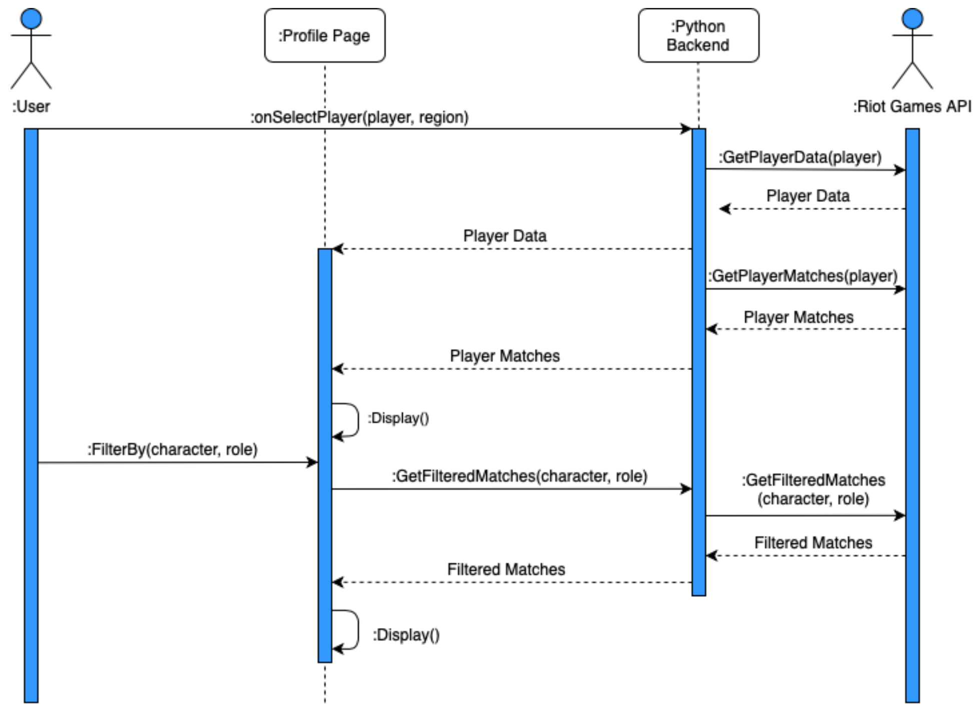


[1] Including comparison of professional player to regular player.

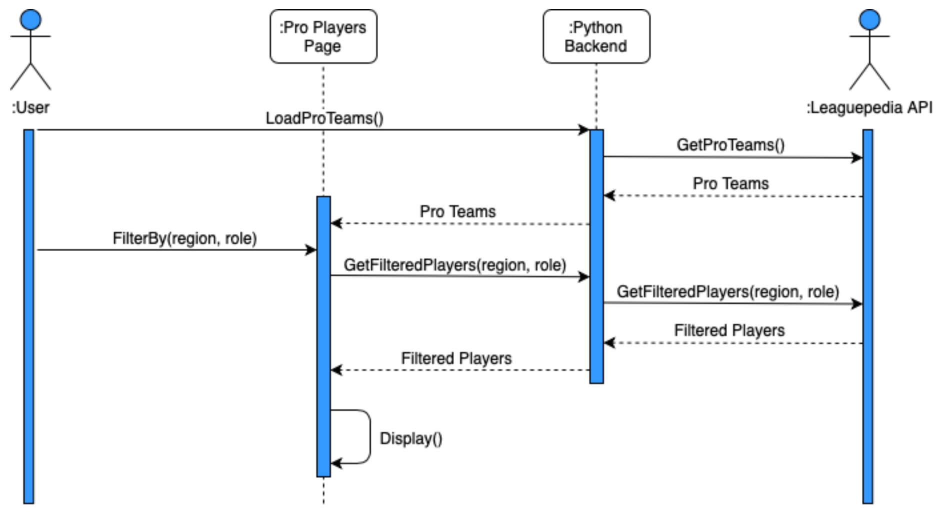
View list of active professional teams and players.



Filtering a player’s match history by character and/or role played.



Filtering professional players by region or role played.


:User
:Pro Players Page
:Python Backend
:Leaguepedia API
LoadProTeams()
GetProTeams()
Pro Teams
Pro Teams
FilterBy(region, role)
GetFilteredPlayers(region, role)
GetFilteredPlayers(region, role)
Filtered Players
Filtered Players
Display()