Package Version Tracking Webpage for CMSSW Releases and Integration Builds

By:

Nourah Alkhaldi

Supervised By:

Ms. Andrea Valenzuela Ramirez

Mr. Malik Shahzad Muzaffar

Duration:

1st of July – 23rd of August

CMS Core Software

CERN

Geneva, Switzerland

Summer 2024

Abstract

The CMS experiment produces a large amount of data that must be stored and analyzed by physicists. The CMS Core Software team works mainly on CMSSW, which is a complex framework designed for data processing, simulation, and analysis of data collected by the CMS detector at CERN. It comprises over 600 packages, including external and data packages. CMSSW IBs are built twice a day at 11:00 and 23:00, and software releases are done on an almost bi-weekly basis. It is crucial to monitor the version of the packages used in each build and to have the capability to compare them across different IBs and releases. The goal of this project is to create an interactive web application that allows users to display the packages information based on their selection criteria (such as package name, platform, architecture, etc.) and to compare the packages in different IBs/releases. Having this information presented in a user-friendly manner would significantly aid both users and release managers of the CMSSW framework.

Keywords**:** CMS, CMSSW, Integration builds, Python, React JS, Releases

Acknowledgements

First and foremost, I would like to extend my deepest gratitude and thanks to my project supervisor Ms. Andrea Valenzuela Ramirez for her continuous support and motivation during the project. Thank you to my family and friends for their constant encouragement. Finally, thank you to Kuwait's Foundation for the Advancement of Science (KFAS) for sponsoring my summer studentship at CERN.

Table of Contents

List of Abbreviations

API: Application Programming Interface

CERN: European Council for Nuclear Research

CI: Continuous Integration

CMS: Compact Muon Solenoid

CMSSDT: CMS Software Development Tools

CMSSW: CMS Offline Software

IBs: Integration Builds

JS: JavaScript

JSON: JavaScript Object Notation

pip: Preferred Installer Program

PyPI: Python Package Index

URL: Unified Resource Locator

Introduction

The CMS experiment produces a large amount of data that must be stored and analyzed by physicists. The CMS Core Software team works mainly on CMSSW, which is a complex framework designed for data processing, simulation, and analysis of data collected by the CMS detector at CERN. CMSSW is mainly written in C++ and Python and is open-source and hosted on GitHub [1]. It comprises over 600 packages, including external and data packages. CMSSW IBs are built twice a day at 11:00 and 23:00, and software releases are done on an almost bi-weekly basis. It is crucial to monitor the version of the packages used in each build and to have the capability to compare them across different IBs and releases.

Project Objectives

The goal of this project is to create an interactive web application that allows users to display the packages information based on their selection criteria (such as package name, platform, architecture, etc.) and to compare different IBs/releases. The current process already extracts a significant amount of useful information from the build process. This project involves evaluating the current methodology, extracting the desired information and developing a web page to display it. Having this information presented in a user-friendly manner would significantly aid both users and release managers of the CMSSW framework.

Requirements and Methodology

There are two main requirements in our project, namely, to extract and parse the IBs/Releases information and display them in a user-friendly manner that includes searching, filtering, and comparing them. The extraction of data can be broken down into two tasks: parsing the IBs/Releases folders and extracting the package details for each IB/Release. We used the Python programming language to execute the folder parsing task, and the Shell/Bash language to execute the data extraction task. As for the interactive web application, we used the React JS library [2]. React allows developers to build fast and dynamic websites with the ability to integrate them with external packages. To integrate the backend Python script with the React project, we created a Flask web server to handle all API requests to the backend. Flask [3] is a web application framework used to handle server requests to a Python backend. Table 1 showcases the project's main requirements. Table 2 details all the different types of languages and tools used in this project and their usage to fulfill the project's requirements.

Table 1: The project's main requirements

No.	Requirement
1	The user should be able to view each IB's information (flavors, build dates, architectures, and packages).
2	The user should be able to view each release's information (architectures, and packages).
3	The user should be able to display the name and version of all the packages used in each IB/Release.
4	The user should be able to click on the package name and be redirected to another page with more information.
5	The user should be able to search for specific packages and their versions (including wildcards) on the website and view the IBs or Releases that use that package and version.
6	The user should be able to compare the package name and version of two IBs/Releases.

Table 2: The different types of languages and tools used in the project and their usage

Type	Name	Usage
Backend	Python	Handling all backend requests and connections to ElasticSearch.
	Flask	Handling API requests.
	Shell/Bash	Extracting the data from the CMSSW folders.
	ElasticSearch	Indexing all IBs/Releases data for faster searching and filtering.
Frontend	React JS	The user interface of the website.
	Material UI	JavaScript package that provides user-friendly UI components.

Implementation

The backend of our project consists of Python functions running on a Flask server. For every new IB, a new folder containing structured data is automatically created by the existing infrastructure. The folder and its subfolders are then parsed to extract the data of the IB, and its packages as shown in the *parse_IB_folder_name()* function in Figure 1. The parent folder's name is structured as follows:

CMSSW_version_flavor_X_date-time

Note that if the flavor part is just 'X', we replace it with the 'DEFAULT' keyword.

Figure 1: IBs folder parsing function

```python
# Function that parses the folder's name and saves each part in a variable
def parse_IB_folder_name(folder_name):
    match = re.match(
        r"^(CMSSW_\d+_\d+)(_?[^_]+)?(_X)?_(\d{4}-\d{2}-\d{2}-\d{4})$", folder_name
    )
    if not match:
        print(f"Folder name '{folder_name}' doesn't match the expected pattern")
        return None
    version = match.group(1)
    flavor_part = match.group(2) if match.group(2) else ""
    flavor = flavor_part.lstrip("_")
    if not flavor:
        flavor = "DEFAULT"
    elif flavor == "X":
        flavor = "DEFAULT"
    date = match.group(4)
    return version, flavor, date
```

After parsing the folders and extracting each field, we traverse through the subdirectories to find the CMSSW IBs JSON file that contains all the packages used in the specific IB along with all its information as shown in the *find_cmssw_ib_file()* function in Figure 2. After we locate the JSON file, we extract all the packages and their information as shown in the *extract_packages()* function in Figure 3. We then create a dictionary containing the IB information with a unique ID for each IB as follows:

CMSSW_version_flavor_date-time_architecture

Figure 2: Finding the CMSSW IBs JSON file to locate the packages information

```
# Recursive function to find the cmssw-ib.json file in deeply nested directories
def find_cmssw_ib_file(start_dir):
    for root, dirs, files in os.walk(start_dir):
        for file in files:
            if file.endswith("cmssw-ib.json"):
                return os.path.join(root, file)
    return None
```

Figure 3: Package information extraction from the JSON file

```
# Function that reads package names and versions from a JSON file
# For both IBs and Releases
def extract_packages(package_file):
    with open(package_file, "r") as f:
        package_data = json.load(f)

    # Dictionary to store package name and version pairs
    package_dict = {}
    for package_key in package_data:
        package_info = package_data[package_key]
        package_name = package_info["name"]
        full_version = package_info["version"]

        # Split the version to remove the checksum
        version_without_checksum = full_version.split("-")[0]

        package_dict[package_name] = version_without_checksum

    return package_dict
```

All the previously mentioned functions are called in the *extract_parse_index_IBs_folders()* function shown in Figure 4, where the user is required to provide a valid path to find all the IBs folders after which the data is to be indexed.

Figure 4: IBs folders parsing, extracting, data dictionary creation, and indexing

```python
def extract_parse_index_IBs_folders(directory):
    result = {}  # Dictionary to hold all results
    # output_file ="ibs_summary.json"
    for item in os.listdir(directory):
        item_path = os.path.join(directory, item)
        if os.path.isdir(item_path):
            parsed_result = parse_IB_folder_name(item)
            if parsed_result:
                version, flavor, date = parsed_result
                for sub_item in os.listdir(item_path):
                    sub_item_path = os.path.join(item_path, sub_item)
                    if os.path.isdir(sub_item_path):
                        package_file = find_cmssw_ib_file(sub_item_path)
                        if package_file:
                            packages = extract_packages(package_file)
                            IB_id = f"{version}_{flavor}_{date}_{sub_item}"

                            current_timestamp = datetime.now().isoformat()

                            result[IB_id] = {
                                "version": version,
                                "flavor": flavor,
                                "date": date,
                                "architecture": sub_item,
                                "packages": packages,
                                "@timestamp": current_timestamp,
                            }
                            doc = result
                            resp = client.index(
                                index="cmssw-ibs", id=IB_id, document=doc[IB_id]
                            )

                            resp = client.get(index="cmssw-ibs", id=IB_id)

                            client.indices.refresh(index="cmssw-ibs")

    return result
```

To fetch each package's information from different package managers, namely, Yum, PyPI, and pip, we created a Shell/Bash script. These package managers provide a central repository to search for packages by keywords and filters. The logic behind the script lies within the package name. If the package's name starts with "py" or "py3-", that means it is a Python package and we need to use either the pip or the PyPI commands to fetch its information as shown in the *get_pip_info()* and *get_pypi_info()* functions in Figure 5. Otherwise, we use the generic Yum command to fetch the desired data as shown in the *get_yum_info()* function in Figure 6.

Figure 5: Bash/Shell script to fetch the Python packages information

```
# Function to get pip package info
get_pip_info() {
  local package_name=$1
  local actual_package_name=$2
  local pip_output

  pip_output=$(pip show "$actual_package_name" 2>&1)

  if echo "$pip_output" | grep -i "WARNING: Package(s) not found:"; then
    get_pypi_info "$package_name" "$actual_package_name"
  else
    local  summary license url
    summary=$(echo "$pip_output" | grep -E "^Summary" |  cut -d: -f2- | xargs)
    license=$(echo "$pip_output" | grep -E "^License" | cut -d: -f2- | xargs)
    url=$(echo "$pip_output" | grep -E "^Home-page" | cut -d: -f2- | xargs)

    info=$(jq -n \
      --arg summary "$summary" \
      --arg license "$license" \
      --arg url "$url" \
      '{ summary: $summary, license: $license, URL: $url}')

    update_json "$pip_details_file" "$actual_package_name" "$info"
  fi
}
# Function to get PyPi package info using curl
get_pypi_info() {
  local package_name=$1
  local actual_package_name=$2
  local pypi_output

  pypi_output=$(curl -s "https://pypi.org/pypi/$actual_package_name/json")

  if echo "$pypi_output" | grep -i '"message": "Not Found"'; then
    echo "No information found for package: $package_name"
    update_json "$no_details_file" "$package_name" '"No matching packages to list"'
  else
    local description summary license url
    description=$(echo "$pypi_output" | jq -r '.info.description' | sed ':a;N;$!ba;s/\n/ /g')
    summary=$(echo "$pypi_output" | jq -r '.info.summary')
    license=$(echo "$pypi_output" | jq -r '.info.license')
    url=$(echo "$pypi_output" | jq -r '.info.home_page')

    info=$(jq -n \
      --arg description "$description" \
      --arg summary "$summary" \
      --arg license "$license" \
      --arg url "$url" \
      '{description: $description, summary: $summary, license: $license, URL: $url}')
    update_json "$pypi_details_file" "$package_name" "$info"
  fi
}
```

For Releases, a similar approach was taken except that the folder path structure for releases is different and thus a different function called *parse_releases_path()* is created as shown in Figure 7. After extracting the release name, release cycle, architecture, and flavor, we extract the packages from the JSON files. Next, we create a dictionary with all the extracted information for each release with a unique ID for each release as follows, where release name is a concatenated field of both the release cycle and flavor:

CMSSW_releaseName_architecture

Figure 6: Bash/Shell script to fetch the packages information using Yum

```bash
# Function to get yum package info
get_yum_info() {
  local package_name=$1
  local yum_output

  yum_output=$(yum info "$package_name" 2>&1)

  if echo "$yum_output" | grep -i "error"; then
    echo "No information found for package: $package_name"
    get_pip_info "$package_name" "$package_name"
    # update_json "$no_details_file" "$package_name" '"No matching packages to list"'
  else
    local description summary license url
    description=$(echo "$yum_output" | awk '/^Description/ {flag=1} /^Summary|License/ {flag=0} flag {print}' | cut -d: -f2- | xargs)
    summary=$(echo "$yum_output" | grep -m 1 -E "^Summary" | cut -d: -f2- | xargs)
    license=$(echo "$yum_output" | grep -m 1 -E "^License" | cut -d: -f2- | xargs)
    url=$(echo "$yum_output" | grep -m 1 -E "^URL" | cut -d: -f2- | xargs)

    info=$(jq -n \
      --arg description "$description" \
      --arg summary "$summary" \
      --arg license "$license" \
      --arg url "$url" \
      '{description: $description, summary: $summary, license: $license, URL: $url}')

    update_json "$output_file" "$package_name" "$info"
  fi
}
```

To create a smooth user experience when searching the website's contents, we indexed all the IBs and Releases data on ElasticSearch [4]. ElasticSearch is an open-source tool that provides data indexing and searching APIs. The first step to achieve our goal of data indexing was to create an account on ElasticSearch that comes with a deployment and a cloud ID. We then created a new API key to be able to access our cloud instance in our backend code.

For IBs, after creating a dictionary of the parsed data, we send it to ElasticSearch to be indexed with the following unique ID for each IB:

CMSSW_version_flavor_date-time_architecture

We also add a timestamp field of the current time and append it to each IB's dictionary before sending it to ElasticSearch. The timestamp field is used by ElasticSearch to create dashboards and discover patterns in the data. Figure 8 shows the *process_and_index_releases_directory()* function that is called to parse the folders, extract the fields and packages, and index the data on ElasticSearch.

Figure 7: Releases directory path parsing

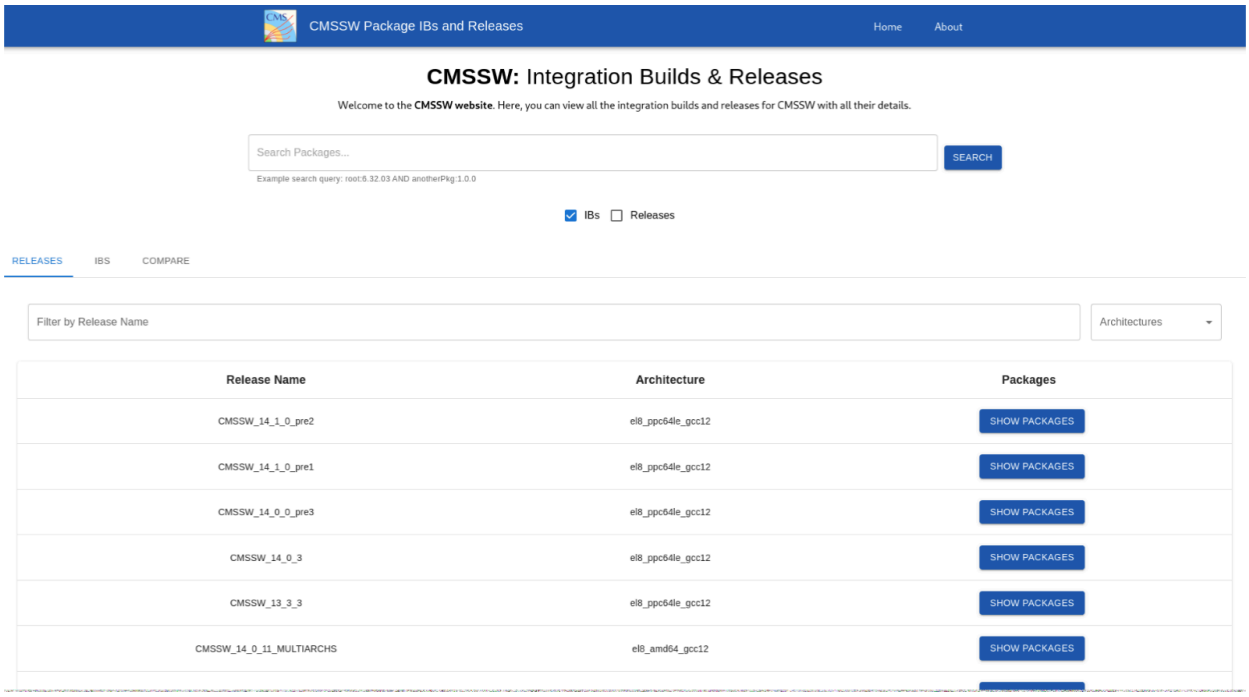
```python
def parse_releases_path(path):
    # Extract Architecture using regex
    architecture_pattern = r"/cms/([^/]+)/"
    # The Version pattern is after the last "/" and before ".json"
    version_pattern = r"/([^/]+)\.json$"

    # Find Architecture
    architecture_match = re.search(architecture_pattern, path)
    if architecture_match:
        architecture = architecture_match.group(1)
    else:
        architecture = "Not found"

    # Find and refine Version
    version_match = re.search(version_pattern, path)
    if version_match:
        full_version = version_match.group(1)
        # Extract the CMSSW version part and any suffix
        version_match = re.search(r"(CMSSW_\d+_\d+(_\d+)*)(_(.*))?", full_version)
        if version_match:
            release_cycle = version_match.group(1)
            flavor = version_match.group(4) if version_match.group(4) else ""
            release_name = f"{release_cycle}_{flavor}" if flavor else release_cycle
        else:
            release_cycle = "Not found"
            flavor = ""
            release_name = "Not found"
    else:
        release_cycle = "Not found"
        flavor = ""
        release_name = "Not found"
    return architecture, release_name, release_cycle, flavor
```

Figure 8: Releases folders parsing, extracting, data dictionary creation, and indexing

```python
def process_and_index_releases_directory(directory):
    releases_info = {}

    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(".json"):
                file_path = os.path.join(root, file)

                # Extract architecture, release_name, release_cycle, and flavor from path
                architecture, release_name, release_cycle, flavor = parse_releases_path(
                    file_path
                )

                if architecture != "Not found" and release_name != "Not found":
                    # Extract packages from JSON file
                    packages = extract_packages(file_path)

                    release_id = f"{release_name}_{architecture}"

                    current_timestamp = datetime.now().isoformat()

                    # Store the data in the dictionary
                    releases_info[release_id] = {
                        "release_name": release_name,
                        "flavor": flavor,
                        "release_cycle": release_cycle,
                        "architecture": architecture,
                        "timestamp": current_timestamp,
                        "packages": packages,
                    }
                    doc = releases_info
                    resp = client.index(
                        index="cmssw-releases", id=release_id, document=doc[release_id]
                    )
                    print(resp["result"])

                    resp = client.get(index="cmssw-releases", id=release_id)
                    print(resp["_source"])

                    client.indices.refresh(index="cmssw-releases")

    return releases_info
```

For the frontend of the project that was created using React JS, and as seen in Figure 9, the Homepage of the website provides a menu with tabs for both IBs and Releases where users can view them in reverse chronological order. Both data are fetched from ElasticSearch using the *all_ibs_index()* and *all_releases_index()* functions shown in Figure 10. The Homepage also contains a search field that searches the indexed data on ElasticSearch for IBs or releases that use packages with specific versions. This will make it easier for users to find the desired data without having to traverse through multiple pages or directories.

Figure 9: The website's homepage



For the IBs tab, as shown in Figure 11, it consists of a table of IB names and architectures. The IB names are created by concatenating the IB's version, build date, and flavor and are displayed in a reverse chronological order. The choice of reversing the order of data was made to make sure that the newer and more relevant data is easier for the user to access. There is also the "Show Packages" button that, when clicked, redirects the user to view and search all the packages and their version that are used in that specific IB as shown in Figure 12. Once a user clicks on the "More Info" button, they are redirected to another page to view that specific package's description, URL, license, and summary as shown in Figure 13.

For the Releases tab, shown in Figure 14 it is created in a similar fashion to IBs except that the release name consists of the release cycle and flavor only. Both the IBs and Releases tabs contain a search by name and a filter by architecture fields to make the user's experience smooth and easy.

Figure 10: Fetch all IBs and Releases data from ElasticSearch

```python
# Function to search the IBs index on ElasticSearch - query obtained from frontend POST request
@app.route("/allIBs", methods=["POST"])
def all_ibs_index():

    # Perform the search query on Elasticsearch
    response = client.search(index="cmssw-ibs", size=10000, query={"match_all": {}})

    # Extract the hits from the responsel
    hits = response["hits"]["hits"]
    results = [hit["_source"] for hit in hits]

    return jsonify(results)

# Function to search the releases index on ElasticSearch - query obtained from frontend POST request
@app.route("/allReleases", methods=["POST"])
def all_releases_index():

    # Perform the search query on Elasticsearch
    response = client.search(
        index="cmssw-releases", size=10000, query={"match_all": {}}
    )

    # Extract the hits from the responsel
    hits = response["hits"]["hits"]
    results = [hit["_source"] for hit in hits]

    return jsonify(results)
```

The third tab in the homepage is the compare tab where users choose IBs and/or Releases from a dropdown menu then compare the versions of the packages used in both choices. It is shown in Figure 15. This feature makes it easier for the users to view and compare two IBs or Releases or a mix of both on one page.

For the search field in the Homepage, the user can search for IBs or Releases that use packages with a specific version in the following format:

$$packageName_1{:}packageVersion_1 \ AND \ packageName_2{:}packageVersion_2 \ AND \ packageName_i{:}packageVersion_i$$

The format also allows for wildcards to expand the search results. Depending on which checkbox is checked, the search button would then make a POST API request to either the *search_ibs_index()* or *search_releases_index()* functions in the backend as shown in Figure 16. Afterwards, the user can view the search results on a separate page as shown in Figure 17.

For both IBs and Releases, the “Show Packages” button redirects the user to a page that displays all the packages used in that specific IB or Release sorted alphabetically as shown in Figure 12.

The results of this project were presented in two meetings with the CMS Core Software group. The CMSSW contributors agreed on the format presented and their comments were incorporated in the final result.

Figure 11: IBs list

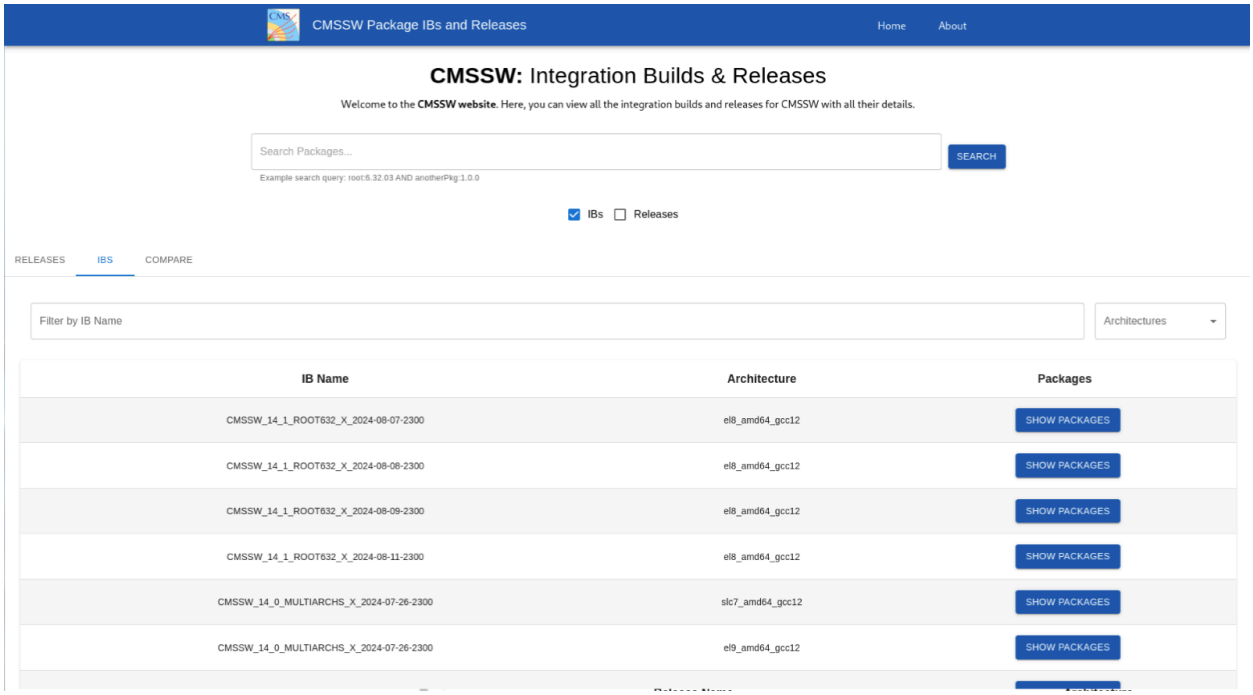


Figure 12: List of packages and their versions

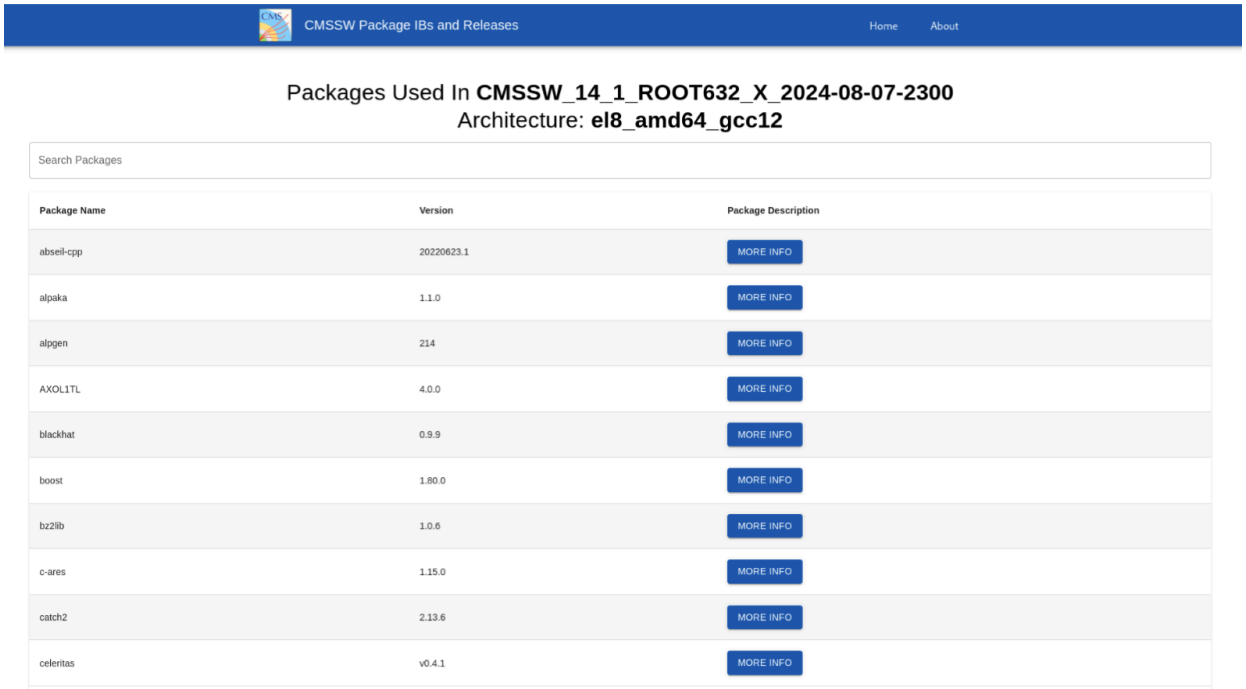


Figure 13: Package information

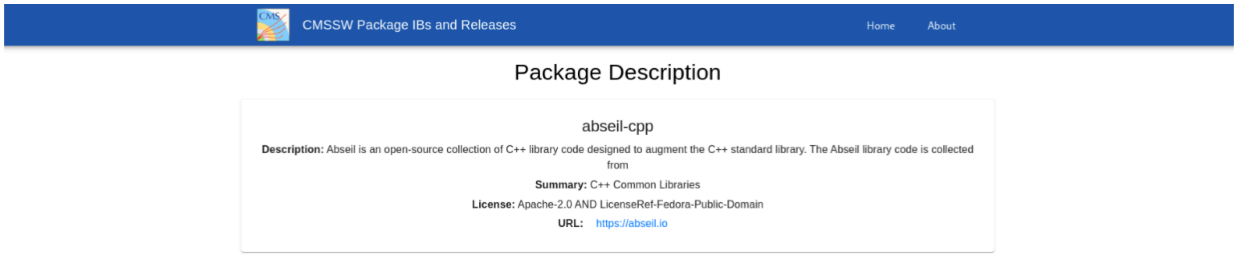


Figure 14: Releases list

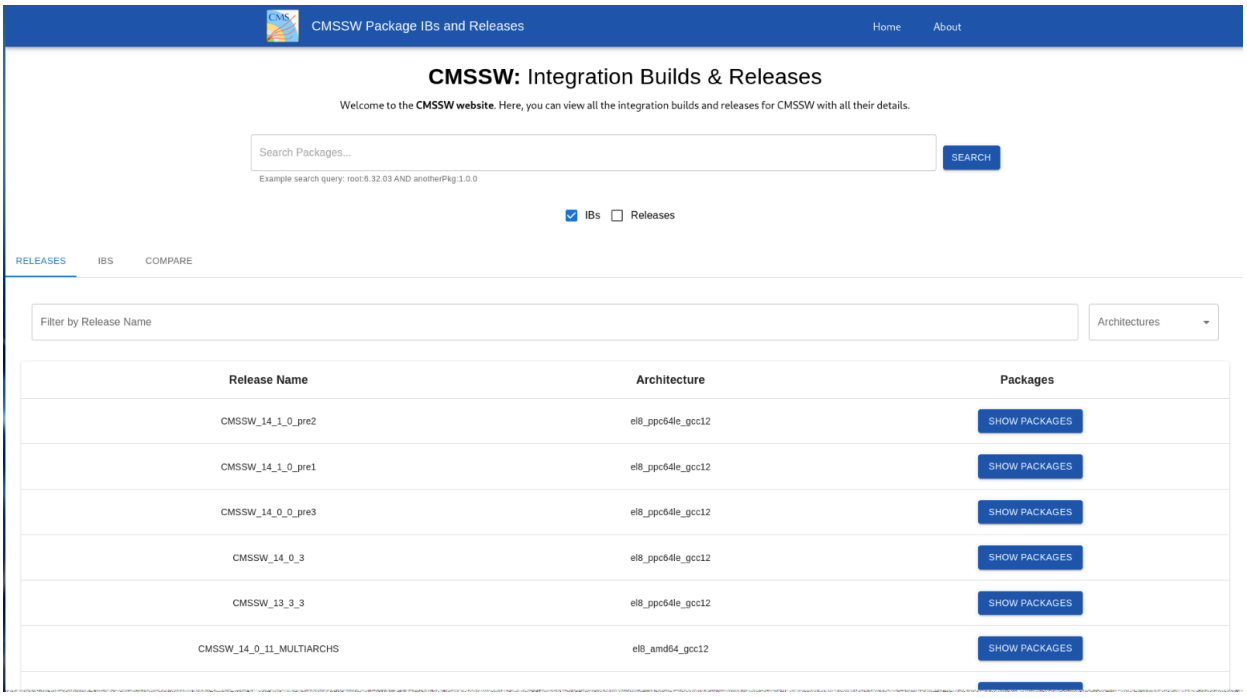


Figure 15: Compare IBs/Releases

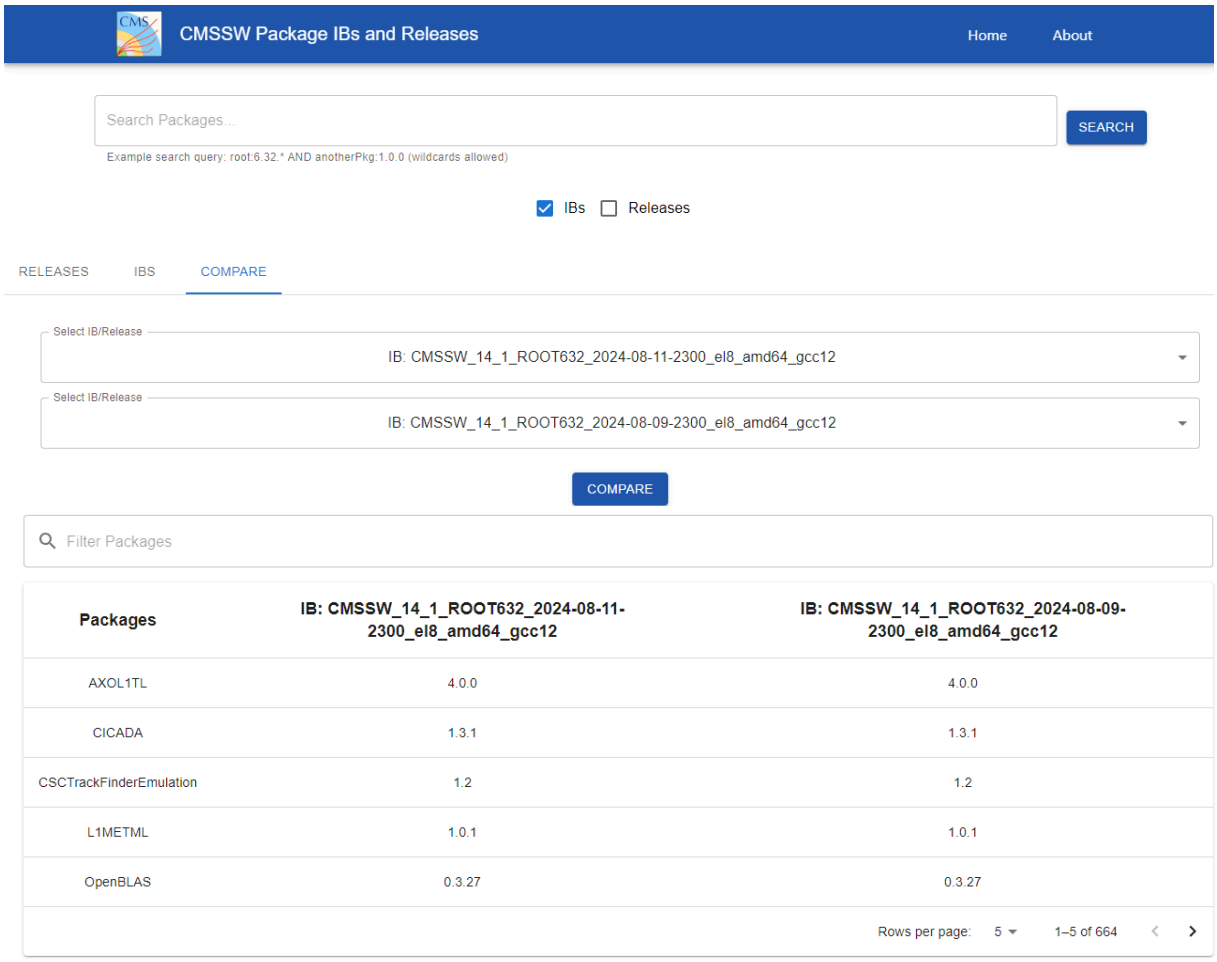
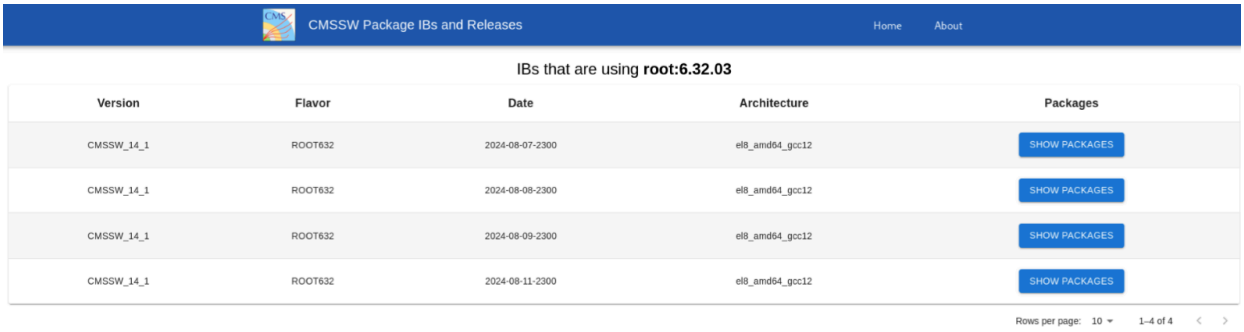


Figure 16: ElasticSearch queries for IBs and Releases for specific packages and versions

```python
@app.route("/searchIBs", methods=["POST"])
def search_ibs_index():
    search_data = request.json
    packages = search_data.get("packages", [])

    # Build Elasticsearch query
    must_clauses = []
    for package in packages:
        package_name = package.get("packageName")
        version = package.get("version")

        if package_name and version:
            must_clauses.append({"term": {f"packages.{package_name}.keyword": version}})

    query = {"query": {"bool": {"must": must_clauses}}}

    # Perform the search query on Elasticsearch
    response = client.search(index="cmssw-ibs", size=10000, body=query)

    # Extract the hits from the response
    hits = response["hits"]["hits"]
    results = [hit["_source"] for hit in hits]

    return jsonify(results)


@app.route("/searchReleases", methods=["POST"])
def search_releases_index():
    search_data = request.json
    packages = search_data.get("packages", [])

    # Build Elasticsearch query
    must_clauses = []
    for package in packages:
        package_name = package.get("packageName")
        version = package.get("version")

        if package_name and version:
            must_clauses.append({"term": {f"packages.{package_name}.keyword": version}})

    query = {"query": {"bool": {"must": must_clauses}}}

    # Perform the search query on Elasticsearch
    response = client.search(index="cmssw-releases", size=10000, body=query)

    # Extract the hits from the response
    hits = response["hits"]["hits"]
    results = [hit["_source"] for hit in hits]

    return jsonify(results)
```

Figure 17: Search results



Conclusion

This project focused on creating a dynamic and user-friendly webpage for CMSSW IBs and Releases. The project's goal and objectives were fulfilled by creating a website that provides that user with features such as searching, filtering, viewing information and comparing. The user interface was created using React JS due to its dynamic nature while the backend was created using Python, Flask, and Shell/Bash. Other tools such as ElasticSearch for data indexing and faster searching were used.

The project's frontend and backend repositories are open-sourced and available on GitHub [5][6]. The CMS Core Software group will work on deploying this project live at https://cmssdt.cern.ch/pkginfo to make it easier for users to access the webpage.

References

[1] CMS Offline Software, https://github.com/cms-sw/cmssw

[2] React JS Library, https://react.dev/

[3] Flask, https://flask.palletsprojects.com/en/3.0.x/

[4] ElasticSearch, https://www.elastic.co/elasticsearch

[5] Frontend code, https://github.com/norahalk/ib-viewer

[6] Backend code, https://github.com/norahalk/backend-scripts