BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PRODUCT DATA VISUALIZATION IN PNC BUILD SYSTEM

VIZUALIZÁCIA PRODUKTOVÝCH DÁT V BUILD SYSTÉME PNC

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PATRIK KORYTÁR**
AUTOR PRÁCE

SUPERVISOR **Ing. RADEK KOČÍ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148505

Institut:	Department of Intelligent Systems (UITS)
Student:	**Korytár Patrik**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Product Data Visualization in PNC Build System**
Category:	User Interfaces
Academic year:	2022/23

Assignment:

1. Study an open-source build system PNC, React framework, and available libraries for data visualization.
2. Analyze product-related pages and users' requirements for product data visualization.
3. Design a new web UI components to improve product data visualization and provide a suitable way to analyze available data.
4. Implement the solution using React framework and contribute the work to the PNC project.
5. Create a video to demonstrate the implemented features.

Literature:

- Adam Wathan, Steve Schoger. Refactoring UI. 2019.
- Steve Krug. Don't Make Me Think. New Riders. 2013.
- ReactJS: Getting started. Online, https://reactjs.org/docs/getting-started.html, září 2022.

Requirements for the semestral defence:
Points 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Kočí Radek, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	3.11.2022

Abstract

The thesis aims to enhance the visualization of Product-related data of the PNC build system on the system's new web user interface. Main visualization elements include tables, charts, network graphs and dashboards. The work analyses PNC user inputs to consider the actual needs of the users. Based on the analysis, completely new UI components were designed and some of the original ones were redesigned. To illustrate the new design, wireframes were created. New REST API endpoints were designed for data needed by the new components. The new features were then implemented. The implementation language is TypeScript. The main libraries include React, Chart.js, Sigma.js and Graphology. As a result, the new PNC system user interface now has new features that help with using the Product-related pages. The implementation is also easily expandable for more visualization of this kind.

Abstrakt

Cieľom tejto bakalárskej práce je zlepšiť vizualizáciu dát súvisiacich s produktami build systému PNC na jeho novom webovom používateľskom rozhraní. Hlavné vizualizačné prvky zahŕňajú tabuľky, grafy, sieťové grafy a palubné panely. Táto práca analyzuje vstupy od používateľov build systému PNC, aby sa zohľadnili skutočné potreby používateľov. Na základe analýzy bol vytvorený dizajn pre úplne nové komponenty používateľského rozhrania a niektoré z už existujúcich boli prepracované. Na ilustráciu nového dizajnu sa vytvorili nákresy. Navrhnuté boli nové koncové body REST API pre dáta vyžadované novými komponentami. Nové funkcie boli následne implementované. Implementačný jazyk je TypeScript. Hlavné knižnice zahŕňajú React, Chart.js, Sigma.js a Graphology. Ako výsledok má teraz nové používateľské rozhranie systému PNC nové funkcie, ktoré pomáhajú s používaním stránok súvisiacich s produktami. Implementácia je tiež ľahko rozšíriteľná pre viac vizualizácií tohto druhu.

Keywords

user interface, visualization, products, artifacts, charts, network graphs, tables, React, Chart.js, Sigma.js, Graphology

Kľúčové slová

užívateľské rozhranie, vizualizácia, produkty, artefakty, grafy, sieťové grafy, tabuľky, React, Chart.js, Sigma.js, Graphology

Reference

KORYTÁR, Patrik. *Product data visualization in PNC build system.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radek Kočí, Ph.D.

Rozšírený abstrakt

Zameranie tejto bakalárskej práce je webové používateľské rozhranie a jeho vylepšenie. Konkrétnejšie, cieľom je zlepšiť vizualizáciu dát súvisiacich s produktmi na novom používateľskom rozhraní build systému PNC. Hlavnými vizualizačnými prvkami sú tabuľky, grafy, sieťové grafy a palubné panely, ktoré zlepšujú vizualizáciu súvislostí a závislostí medzi entitami systému PNC, konkrétne tými, ktoré súvisia s produktmi.

Systém PNC má momentálne dve používateľské rozhrania. Prvé, pôvodné, je napísané v knižnici Angular. Táto pôvodná verzia používateľského rozhrania sa práve prepisuje do knižnice React. Táto bakalárska práca má za cieľ vylepšiť práve toto nové rozhranie oproti tomu pôvodnému.

Počiatok tejto práce spočíval v naštudovaní implementačného jazyka a knižníc. Implementačný jazyk je TypeScript. Najdôležitejšie knižnice sú React pre tvorbu používateľského rozhrania, Chart.js pre tvorbu grafov, Graphology pre tvorbu dátových štruktúr sieťového grafu a Sigma.js pre vizualizáciu sieťových grafov. Tieto boli naštudované predovšetkým z ich dokumentácií dostupných online.

Následne bolo potrebné analyzovať entity systému PNC a vzťahy medzi nimi. Systém PNC slúži na správu a spúšťanie buildov kódu aplikácií. Systém PNC pozostáva z viacerých samostatných aplikácií, ktoré sú rozdelené do verzií. Verzie samotné sa rozdeľujú do míľnikov. Jeden z míľnikov verzie môže byť označený ako súčasný. Predtým, než sa môže vytvoriť build, musí sa najprv vytvoriť jeho konfigurácia. Konfigurácia buildu obsahuje odkaz na kód aplikácie, ktorá sa má preložiť. Taktiež obsahuje skript, podľa ktorého sa má build vykonať. Konfigurácia buildu je prepojená s konkrétnou verziou produktu. Spustená konfigurácia vytvára entitu buildu, ktorá reprezentuje spustený build proces. Build sa prepojí s míľnikom, ktorý je v čase spustenia buildu označený ako súčasný vo verzii, ktorá prislúcha konfigurácii buildu. Výsledkom buildu sú artefakty, čo sú súbory, ktoré je možné stiahnuť. Artefakty môžu slúžiť ako závislosti iných buildov. Artefakty, ktoré boli doručené zákazníkovi v rámci archívu produktu, sa nazývajú doručené artefakty. Doručené artefakty sú viazané na konkrétny míľnik. Archív produktu sa skladá z artefaktov vyprodukovaných v rámci buildov míľniku.

Po naštudovaní teórie nasledovalo spracovanie používateľských požiadaviek ohľadom zlepšenia vizualizácie entít súvisiacich s produktmi. Medzi stránky na pôvodnom používateľskom rozhraní, ktoré bolo treba vylepšiť, patrí stránka tabuľky artefaktov, stránka detailu verzie produktu a stránka detailu míľniku produktu.

Hlavnými nedostatkami stránky artefaktov sú zobrazenie identifikátorov artefaktov, chýbajúce prepojenie s buildom, ktorý vyprodukoval artefakt, a fakt, že veľkú časť tabuľky zaberajú kontrolné súčty. Identifikátory artefaktov sa skladajú z viacerých častí, a bolo by vhodné tieto časti vizuálne zvýrazniť.

Na stránke detailu míľniku produktu sú zobrazené len detaily ohľadom položiek míľniku, ale chýbajú štatistiky a grafy ohľadom prepojenia míľniku s entitami, ktoré mu patria, napríklad štatistiky ohľadom doručených artefaktov. Doručené artefakty míľniku majú rôzne vlastnosti, z ktorých sa dá spraviť graf ich distribúcie. Taktiež je možné spraviť štatistiky o zdroji doručených artefaktov, teda o tom, z akého buildu pochádzajú. Na úrovni verzie produktu je možné spraviť podobné štatistiky a grafy, ktoré predstavujú agregáciu ich ekvivalentov na stránkach detailu míľnikov jednej verzie produktu. Grafy a štatistiky by mohli pretvoriť stránku detailu do palubného panelu.

Komponenty, ktoré boli navrhnuté priamo používateľmi, zahrňujú sieťové grafy zobrazujúce vzťahy a závislosti medzi entitami systému PNC, ako aj komponenta na porovnávanie míľnikov a verzií doručených artefaktov.

Po spracovaní používateľských požiadaviek boli vypracované nákresy dizajnu nových alebo redizajnu starých stránok pôvodného používateľského rozhrania. Na základe týchto nákresov bola vyhotovená implementácia týchto stránok na novom používateľskom rozhraní. Taktiež bolo nutné navrhnúť rozhranie REST API pre koncové body potrebné pre získanie dát pre nové stránky používateľského rozhrania. Koncové body však boli len navrhnuté, ich implementácia nie je súčasťou tejto bakalárskej práce.

Navrhnuté a implementované boli nasledovné stránky: redizajn tabuľky artefaktov, palubný panel míľniku produktu, palubný panel verzie produktu, sieťový graf zdieľaných doručených artefaktov medzi míľnikmi produktu, sieťový graf závislostí medzi buildami a tabuľka porovnania doručených artefaktov medzi míľnikmi produktu. Implementácia je navyše ľahko rozšíriteľná o štatistiky podobného charakteru, napríklad nové sieťové grafy alebo štatistiky týkajúce sa buildov míľniku produktu.

Práca bola spracovaná vo forme Pull Requestov do GitHub repozitára projektu nového používateľského rozhrania systému PNC, celkový počet činí 20 Pull Requestov. Zadanie bolo úspešne splnené.

Product data visualization in PNC build system

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Radek Kočí Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Patrik Korytár
May 10, 2023

Acknowledgements

I would like to thank my supervisor Ing. Radek Kočí Ph.D. for supervising this Bachelor's thesis. Many thanks to Ing. Jakub Barteček for the organization of this Bachelor's thesis, Mgr. Martin Kelnar for guidance with the design of the UI and Mgr. Jan Brázdil for providing information regarding the current PNC backend state and relationships between PNC entities. And I am grateful to my family and friends too.

Contents

List of Figures

Chapter 1

Introduction

This Bachelor's thesis area of focus is the improvement of the web user interface. It is developed in collaboration with the Red Hat company. Improvements are made to the new web user interface (UI) of the PNC build system. Namely, it focuses on the enhancement of visualization of Product-related pages to portray relationships and interconnections between Product-related entities in a more straightforward way.

Currently, PNC uses outdated web UI written in the JavaScript language and the Angular library. This original UI is now being rewritten to the TypeScript language and the React library. This Bachelor's thesis is implemented for the new UI and Product-related pages are visualized in a more enhanced way than it is done on the original UI. The original UI of Product-related pages is mainly composed of rather simple UI components – detail pages, list pages, and create and update pages. For example, detail of a specific Product or a list of Artifacts. These pages contain data of entities stored directly in database tables. It lacks visualization of inferred relationships and interconnections between the Product-related entities. For example, there is no direct way to learn which Artifacts are shared between Product Milestones or to see Build dependencies based on Artifacts used in them.

To address this issue, this Bachelor's thesis implements components including tables, charts, network graphs, and dashboards. Dashboards are made up of data cards and charts that display statistics. Network graphs connect entities based on relationships or dependencies between them. Tables list collections of entities or display relationships between different entities in the 2D grid. A secondary goal of this Bachelor's thesis is to create the basis for future additional visualization so that, for example, the display of other statistics can be easily added.

This Bachelor's thesis analyzed user needs to determine what kind of information and interconnections between Product-related entities would be useful to have. Then, these were taken into account during the design process. The newly designed pages need new data; therefore, new REST API endpoints were designed. The new endpoints were not implemented, because it is not the concern of this Bachelor's thesis. Then the implementation was done. The implementation language is TypeScript. The main implementation libraries include React, Chart.js, Sigma.js, and Graphology.

According to the process mentioned above, the Bachelor's thesis is divided into the following chapters. The second chapter explains the theory of the PNC build system and entities from a high-level point of view. Chosen relationships between the PNC entities are shown. The third chapter lists the most important technologies used for the implementation side of this Bachelor's thesis, including the language and libraries. Some concepts of the libraries used are presented. The fourth chapter describes the current state of the Product-

related pages and their problems. Then, user requirements regarding the enhancement of these pages are reported. The fifth chapter illustrates the new design for the UI components that were implemented. The sixth chapter presents the new REST API endpoints that are needed by the new UI components. The seventh chapter dives into implementation details and describes the testing of the work. The eighth and final chapter is the conclusion.

Chapter 2

PNC build system

This chapter explains what a PNC build system is and describes its entities, mainly those that relate to this Bachelor's thesis. It also delves into some existing relationships between the entities that were visualized in this Bachelor's thesis.

When referring to the PNC system entities, their names are capitalized throughout the entire Bachelor's thesis. The same words as are used to name the PNC entities are written in lowercase when the PNC entity is not addressed specifically, but instead, a generic term is meant by the word.

2.1 PNC build system

PNC stands for Project Newcastle. It is a *"system for managing, executing, and tracking builds"* [9]. The purpose of the system is to perform builds of versioned software (products) stored in the repositories. The system allows managing versions of the software and provides ways to analyze builds and artifacts. Artifacts are files created by the builds or used by them. It can be, for example, the code compiled from an application. PNC consists of multiple microservices, but this Bachelor's thesis will not delve into it in any way.

PNC system currently has two web user interfaces (UIs) through which the system is used. The first UI is written in an obsolete version of the Angular JavaScript library. This original UI is now being rewritten to the React library. Through this Bachelor's thesis, the original version will be referred to as the original UI and the new version as the new UI.

2.2 PNC build system entities

This section clarifies the purpose of the PNC entities used in this Bachelor's thesis.

Product

Product is a standalone application or a deliverable package[1]. Product is versioned by Product Versions and Product Milestones.

Build Configuration

Build Configuration or Build Config is a set of parameters that are used when the build process is executed. Build Configuration contains a link to an SCM repository to be built

[1] Information is obtained from internal PNC system documentation.

and a script according to which the build is executed. To manage various versions of the software and its builds, Build Configuration is linked to one Product Version.

Build

Build entity is a record of the build process performed according to its Build Configuration containing the build script and the link to the repository to be built. Build is linked to a current (current at the time of build execution) Product Milestone of a Product Version linked in its Build Configuration. The result of a performed Build is a set of Artifacts. Artifacts can also be dependencies of other Builds.

Product Version

Product Version or Version represents a version of a Product. It is the combination of the major and minor version numbers in the semantic versioning[2] (for example, Product Version 1.3). Build Configurations are linked to it. Product Version is divided into Product Milestones. One of the Product Version's Product Milestones can be marked as current.

Product Milestone

Product Milestone or Milestone represents a subversion of a Product Version. It represents the patch version number in the semantic versioning (for example, Product Milestone 1.3.4). Builds performed according to Build Configurations of some Product Version are linked to that Product Version's current Product Milestone.

Artifact

Artifact is an archive, such as `jar`, `xml` or `tgz` file, that is either produced by a Build (Build result) or used by a Build (Build dependency). It is the smallest unit of dependency of a Build, Product Milestone, Product Version, or a Product.

Artifacts can be produced by building different types of repositories. These include the Maven, NPM, and CocoaPods repositories. Based on the type of repository Artifacts are coming from, Artifacts have different formats of their identifiers (names).

NPM Artifact identifier is divided into two parts separated by a colon. The first part is a package name and the second is a version of the Artifact. The name can be prefixed with a scope. Scope enables packages to be grouped together, for example, a group of packages of a certain organization. The scope starts with the at character (@) and ends with the dash symbol [6, 7].

Example of a PNC NPM Artifact identifier: `abbrv:1.0.1`. NPM Artifact in the PNC system is an archive of an NPM package.

Maven Artifact identifier is divided into the following parts [14, 15].

1. **Group ID** – groups related Artifacts or Artifacts of one organization
2. **Artifact ID** – unique identifier of the Artifact within the Group ID group
3. **Artifact version**

[2] https://semver.org/

4. **Classifier** – optional; used to distinguish between Artifacts of the same name that contain different content; for example, the `sources` classifier is typically used for the source code of the Artifact

In the PNC system, the Maven Artifact identifier has one additional part. After the Artifact ID, the Artifact archive type follows (for example, `jar`). Example of a PNC Maven Artifact identifier that has all five parts: `xml-resolver:xml-resolver:jar:1.2.0:sources`.

There are multiple sets of Artifacts that are related to a Build. The first set, called Used Artifacts in this Bachelor's thesis, contains all Artifacts which were used in a Build and form Build's dependencies. The second set, in this Bachelor's thesis named Produced Artifacts, is a set of all Artifacts produced by a Build. These may be dependencies for other Builds. Delivered Artifacts is a set related to a Product Milestone and is described below.

Deliverable and Delivered Artifact

Information in this section was obtained from internal PNC system documentation.

In short, a Delivered Artifact is an Artifact delivered to a customer found in a Deliverable archive.

Deliverables are archives (such as `zip` files) delivered to customers. A Deliverable archive is a compilation of Artifacts produced by one Build or multiple Builds of a certain Product Milestone. It is up to a PNC user to compile it and then deliver it to a customer. For the compilation of the archive, Build itself can be used to compile Artifacts into the archive, which will be delivered.

Delivered Artifacts is a set of Artifacts linked to one Product Milestone found by the process named Deliverables Analysis which is managed by a PNC microservice called Deliverables Analyzer. Deliverables Analyzer receives URL links to Deliverables archives on its input. These archives are analyzed and Artifacts are found in them. The analyzer tries to find these Artifacts in Red Hat-approved build systems, one of which is PNC. Then these Artifacts are connected to Builds which produced them (if any, the Artifact possibly did not have to be found in any build system). In other words, Deliverables Analysis finds where the content delivered to a customer comes from.

SCM Repository

Source Code Management Repository or SCM Repository or Repository is a git repository where the application code is stored.

2.3 PNC build system entity relationships

This section illustrates selected relationships between the PNC entities.

2.3.1 Product Milestone interconnections

Sharing of Used or Delivered Artifacts between Product Milestones is illustrated in Figure 2.1. The figure shows distinct sources of Artifacts.

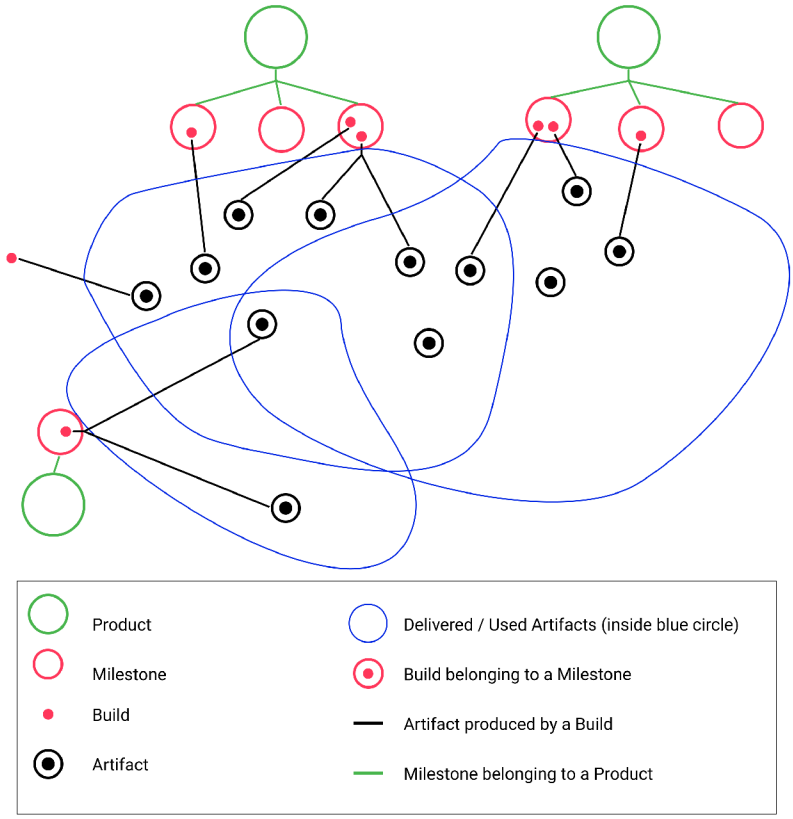

Figure 2.1: Interconnections between Product Milestones in the PNC build system

The blue circle represents a set of Used or Delivered Artifacts of a Product Milestone. In the case of Used Artifacts, the set is a collection of all dependencies of all Builds of a Product Milestone. For Delivered Artifacts, the set represents Artifacts delivered to a customer in a Product Milestone. No matter whether these are Used or Delivered Artifacts, the principle of Artifacts' source and sharing of Artifacts between Product Milestones applies to both of the sets.

Intersections of blue circles are sets of Artifacts shared between the two Product Milestones. That means a set of Artifacts either used by the Builds of both Product Milestones or Artifacts delivered by both Product Milestones.

There are multiple possible sources of an Artifact based on the Build that produced it. The most common is a Build contained in a Product Milestone. The Product Milestone which contains that Build can be the same Product Milestone to which the Delivered or Used Artifacts set belongs. Or it can be a Product Milestone that is part of the same Product Version or at least the same Product as the one to which the set belongs. Or

it can be from a different Product entirely. Some Artifacts are produced by Builds not belonging to any Product Milestone (which is sometimes the case for Builds done in a build system other than PNC). Another state is when Artifact was not produced by any Build because the Artifact was imported into the system.

The same relationships exist on the Product Version or Product level. In those cases, the sets are collections of all Used or Delivered Artifacts of all Product Milestones of the same Product Version or the same Product.

2.4 Build-time dependencies

Build-time dependencies are dependencies that are formed when Build is performed. Figure 2.2 shows an example of build-time dependencies between Builds and Product Milestones. When one Build uses an Artifact produced by another Build, the former Build becomes dependent on the latter Build. The same build-time dependency exists on a Product Milestone level. That means a Product Milestone which contains the dependent Build becomes dependent on a Product Milestone which contains the dependency Build.

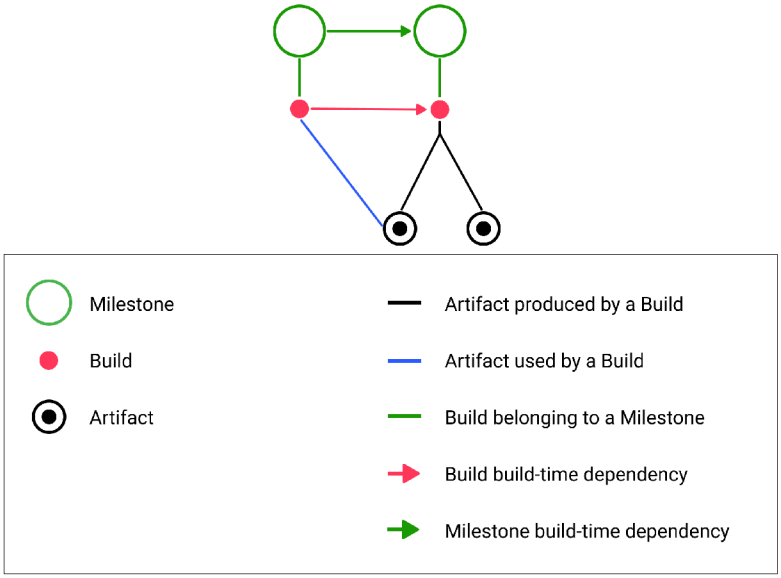

Figure 2.2: Build-time dependencies between Builds and Product Milestones in the PNC build system

Chapter 3

Used technologies

This chapter describes the most important technologies, both languages and libraries, used for the implementation of this Bachelor's thesis. The selected concepts are explained.

3.1 TypeScript

This section was composed with the help of the TypeScript documentation [5].

TypeScript is a typed superset of the JavaScript programming language. TypeScript allows code to be statically typed. "Typed" means that the language adds rules defining how different types of values can be used. "Static" means that type checks are performed before code execution.

TypeScript is a superset of JavaScript; therefore, the valid JavaScript code is syntactically valid TypeScript code. Still, valid JavaScript code may contain semantic errors from the point of view of TypeScript, such as when the variable's value is used incorrectly. For example, the code in Listing 3.1 is valid JavaScript, but invalid TypeScript code because TypeScript aims to catch potential errors.

```
// number two divided by an array
const a = 2 / [];
```

Listing 3.1: TypeScript code containing a semantic error

TypeScript code is compiled into JavaScript code. Simply put, during compilation, type checks are done, and then, if the code is valid, all specified types are erased from the code, the result being JavaScrip code. Therefore, the TypeScript code preserves the run-time behavior of JavaScript. TypeScript does not provide any additional run-time functionality, such as new functions.

3.1.1 Types and interfaces

In many cases, TypeScript will infer types of values and variables. For example, variable `const count = 124;` is automatically detected to be a number.

The other way to determine the types of values and variables is by type annotations. Type annotations assign a type to values, functions (both parameters and return value), or variables. TypeScript offers basic types, including string, number, or boolean. The array is a type that signifies a list of values of the same type. Listing 3.2 demonstrates the type annotations.

```
const name: string = "A1";
// function accepting two numbers and returning their sum
const addNumbers = (x: number; y: number): number => x + y;
const values: number[] = [1, 2, 3]; // number[] => array of numbers
```

Listing 3.2: TypeScript type annotations example

TypeScript also provides a way to declare object types which are composed of primitive types and/or other object types. One of the multiple ways to declare the object type is by interfaces. Interfaces describe the structure of the object and its properties, whether optional or required. The code in Listing 3.3 shows how to declare an interface and apply it on a variable.

```
interface Project {
  id: number;
  name: string;
  version?: string; // ? = optional
}

const project: Project = { id: 12, name: "Random project" };
```

Listing 3.3: TypeScript interface example

TypeScript uses duck typing. This means that the object's shape is taken into account, and when types have the same shape (the same properties), they are considered to be of the same type, despite the different type annotations. In the valid code in Listing 3.4, the given object is passed to the `printPerson` function as its parameter, even though it was not given any type annotation. The code is valid because the function parameter and the object have the same structure.

```
interface Person {
  name: string;
  age: number;
}

const printPerson = (person: Person) => {
  console.log(person.name, person.age);
}

printPerson({ name: "Peter", age: 72 });
```

Listing 3.4: TypeScript duck typing example

3.2 React

This section was written with the help of the React site and documentation [4].

React is a JavaScript and TypeScript library whose goal is to create and render user interfaces. The building block of the user interfaces in React is a component. The component is a piece of a user interface, for example, a button, text, or a page. Components are

reusable, and they can be combined and nested in order to create more complex components.

3.2.1 Components

The component is a building block of the user interface. In React, the component is implemented as a JavaScript function. The component has its own UI logic and visual output (markup). Markup is represented in a syntax extension called JSX. The syntax of JSX is similar to that of HTML.

Components can have a state storing various data, for example, a text in a text input. To remember a value, the `useState` function (hook) is used. `useState` accepts the initial value of the stored value and returns an array containing a state variable and state setter function. The state variable is the stored value. The state setter function updates the stored value.

Listing 3.5 illustrates how to create a state in React.

```
const [count, setCount] = useState(0);
```

Listing 3.5: React state example

React renders a component by calling a function that represents the component. This works recursively. If the component returns some other component, React will also render that component. The component is rendered on its initial render when the root component and its child components were rendered for the first time. The root component is the starting point of the rendering of React components. When the component's state is changed (with the state setter function), the component is re-rendered, and UI is updated accordingly to the new state.

Listing 3.6 displays how to create React components, their state, and nest them.

3.2.2 Hooks

Hooks are means to share the state logic between components. Hooks are implemented as functions. They can have their own states. They implement the logic of those states and return some states. Hooks are called inside components, and states returned by the hooks can be used inside the components. This way the state logic has been extracted to a single place (hook) that can be reused in multiple components.

React offers built-in hooks, `useState` is one of them, but custom hooks can be created. Custom hooks can be used to fetch data or keep track of whether the full-screen mode is active, for example.

3.2.3 useEffect hook

`useEffect` is a built-in hook that is used to synchronize with or connect to external systems. For example, fetching server data, controlling non-React code, or sending logs to the server.

`useEffect` hook accepts a function and an array of dependencies. The array of dependencies is a list of states. The function passed to the hook is executed after the component re-render that is caused by an update of some of those states. The passed function is also executed after the first render of the component. If no array of dependencies is provided, the passed function executes after every render of the component.

```
// onClick, children - properties of a component
// children - child component(s)
const Button = ({children, onClick}) => {
 // rendered UI
 return (
   // {children} - curly braces are used to include JavaScript code in JSX
   <button onClick={onClick}>{children}</button>
 )
}

const Card = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Button has been pressed {count} times.</p>
      <Button onClick={() => setCount(count + 1)}>Press me!</Button>
    </div>
  )
}
```

Listing 3.6: Example of React components and their nesting

Sample in Listing 3.7 demonstrates the `useEffect` hook, which sends a log to the server each time the data are updated.

```
useEffect(() => {
  // if data are non-empty
  if(data) {
    // run function to send a log to the server
    sendLog(data);
  }
}, [data]);
```

Listing 3.7: React useEffect hook example

3.3 PatternFly

PatternFly is an open-source design system providing standards, guidelines, and tools to develop and build user interfaces. It provides UI components which are modular building blocks of the UI. It also offers layouts that help to create the page structure. PatternFly is implemented as a React library providing components to build the UI, although HTML and CSS variants of the components are also provided. Components offered by PatternFly include buttons, selects, forms, flex and grid containers, cards, tables, popovers, and many others [11].

3.4 Chart.js

The source of this section is the Chart.js documentation [1].

Chart.js is a JavaScript library for the creation and rendering of charts. Chart.js offers multiple types of charts, including doughnut charts, bar charts, and line charts. There are also multiple plugins available, and Chart.js charts are customizable. Charts are rendered into an HTML5 canvas[1] unlike some other libraries rendering as SVG. The positive consequence of that is better performance since there are not thousands of SVG nodes created in the Document Object Model[2] tree for large datasets.

3.4.1 Doughnut chart example

An example of a Chart.js doughnut chart configuration is included in Listing 3.8.

```
const config = {
  type: 'doughnut',
  data: {
    labels: [
      'Red'
      'Green',
      'Blue',
    ],
    datasets: [{
      label: 'Dataset1',
      data: [12, 130, 94],
    }]
  }
};
```

Listing 3.8: Chart.js doughnut chart configuration example

`type` in a Chart.js configuration specifies the type of the chart, in this case, it is a doughnut chart. For the doughnut chart, `data` is composed of `labels` and a `dataset`. The `labels` are the names of the segments that display the proportional values of the data. The `dataset` is the set of proportional values of the data. The `dataset` is composed of its `label` and an array of values of that dataset. In the provided example, 12 corresponds to the "Red" label, 130 to the "Green" label, and 94 to the "Blue" label.

In the Document Object Model, a canvas element must exist into which the chart will be rendered. Then, with reference to the canvas and the chart configuration, the chart can be created and rendered, as can be seen in Listing 3.9.

```
// 'example-chart' = id of the canvas
const ctx = document.getElementById('example-chart');
const chart = new Chart(ctx, config);
```

Listing 3.9: Chart.js chart creation example

[1] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
[2] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

3.5 Graphology

This section was written with information extracted from the documentation of the Graphology library [8].

Graphology is a JavaScript and TypeScript library that provides a multipurpose data structure for a network graph. It supports various kinds of graphs, including directed or undirected graphs, and all types of graphs share the same interface. The library provides methods to manipulate, traverse, and analyze the graph, such as adding or removing nodes and edges. The library also has multiple extending libraries offering graph theory algorithms and utilities, such as graph layouts. The entire graph is represented by a single `Graph` object.

To be more specific, Graphology is a specification for a library, but a reference implementation is also proposed. The reference implementation is used in this Bachelor's thesis.

3.5.1 Introduction to the Graphology library

The graph is created by instantiating the `Graph` object. The node is added to the graph using the `addNode` method and the edge is created using the `addEdge` method. The `addNode` method accepts the name of the node and an optional object of the node's attributes, for example, the node's label. In the graph, the node is referred to by its name assigned to it at its creation. Arguments passed to the `addEdge` method are names of two nodes and optionally an object of edge's attributes. All edges have a key, and the `addEdge` method automatically generates it. `addEdgeWithKey` method allows an edge to be created with a specific key. The nodes and edges can be deleted using the `dropNode` and `dropEdge` methods. `clear` method deletes all nodes and edges from the graph. Listing 3.10 portrays how to create the graph in Graphology.

```
const graph = new Graph();
graph.addNode('BuildA', { label: 'Build A' });
graph.addNode('BuildB', { label: 'Build B' });
// edge interconnecting BuildA and BuildB
graph.addEdge('BuildA', 'BuildB', { sharedArtifacts: 4 });
graph.clear();
```

Listing 3.10: Example of creation of the Graphology graph

`hasNode` method returns a boolean whether the passed node exists in the graph. `hasEdge` method returns whether there is an edge that connects two passed nodes. To find out whether some edge is connected to a node, `hasExtremity` can be used. `extremities` method returns the nodes that are interconnected by the passed edge. Listing 3.11 demonstrates some of these methods.

```
const edge = graph.addEdge('BuildC', 'BuildD');
// returns true
graph.hasEdge('BuildC', 'BuildD');
// returns true
graph.hasExtremity(edge, 'BuildD');
// returns ['BuildC', 'BuildD']
graph.extremities(edge);
```

Listing 3.11: Example of reading Graphology methods

To set a node attribute to a specific value, `setNodeAttribute` can be used to create a new attribute or update an old one. `updateNodeAttribute` does not set an attribute to a specific value, but accepts a callback which accepts the current value of the attribute and returns the new one. `getNodeAttribute` method returns the specific attribute of the node. `getNodeAttributes` returns all attributes of the node. The same set of methods exists also for edges – `setEdgeAttribute`, etc. An example is shown in Listing 3.12.

```
graph.setNodeAttribute('BuildA', 'x', 500);
// increase 'x' by one
graph.updateNodeAttribute('BuildA', 'x', x => x + 1);
// returns 501
graph.getNodeAttribute('BuildA', 'x');
// returns the object of all node attributes
graph.getNodeAttributes('BuildA');
```

Listing 3.12: Example of node attribute methods in the Graphology library

Graphology offers multiple methods to iterate over a set of nodes and edges. Some of them accept a callback that is executed for each node or edge of some set. This is the case for, for example, the following methods. `forEachNode` iterates over each node of the graph. `forEachEdge` iterates over all edges of the graph. `forEachNeighbor` iterates over neighbors of the passed node, and a demonstration of this is presented in Listing 3.13.

```
// prints neighbors of the node 'BuildA'
graph.forEachNeighbor('BuildA', (neighbor) => {
  console.log(neighbor);
});
```

Listing 3.13: Example of iteration over neighbors of a node in the Graphology library

3.5.2 ForceAtlas2 layout algorithm

Graphology offers extending libraries implementing graph layout algorithms. One of them is ForceAtlas2.

ForceAtlas2 is a graph layout algorithm that simulates a physical system of attractive and repulsive forces to position nodes and edges. The nodes are repulsed by each other, but the edges attract their nodes [2].

The code in Listing 3.14 displays the Webworker variant of the ForceAtlas2 layout and its usage.

```
const layout = new FA2Layout(graph, {
  settings: { gravity: 1 }
});

// start the layout
layout.start();
// stop the layout
layout.stop();
// is the layout running?
layout.isRunning();
```

Listing 3.14: ForceAtlas2 Graphology example (Webworker implementation)

Webworker is used to run scripts in background threads that perform tasks without interfering with the user interface [3]. Graphology also has a synchronous variant of the layout.

3.6 Sigma.js

Sigma.js does not provide comprehensive documentation. This section was composed by means of help from the Sigma.js GitHub repository, including code and README files contained in the repository [13].

Sigma.js is a JavaScript library that aims to visualize network graphs. It manages the graph rendering and interaction with it. The graph data structure is managed by the Graphology library. For rendering, Sigma.js uses WebGL. Sigma.js is accustomed to the rendering of thousands of nodes and edges.

3.6.1 Introduction to the Sigma.js library

An example of the creation of the Sigma.js network graph is displayed in the code in Listing 3.15.

```
const sigmaDiv = document.getElementById('sigma-div');
// graph - Graphology graph
const renderer = new Sigma(graph, sigmaDiv, { /* settings */ });
```

Listing 3.15: Sigma.js graph creation example

Graphology graph must be created first. Also, the container to which the graph will be rendered must exist in the Document Object Model, for example, a div container. Then the Sigma object can be instantiated. The Sigma object represents the renderer of the graph.

The nodes and edges of the graph are rendered on the basis of the attributes of the nodes and edges. These include the sizes, labels, colors, and coordinates of the nodes or edges. Listing 3.16 provides a code adding a node and an edge to the graph with these attributes specified.

The example node will display the “Build A” label, its size will be 6, its color black, and it will have a set position. The example edge will be of size 5 and black in color.

```
graph.addNode('BuildA', {
  label: 'Build A',
  size: 6,
  color: 'black',
  x: 45,
  y: 30
});

/* BuildB is added too */

graph.addEdge('BuildA', 'BuildB', {
  size: 5,
  color: 'black'
});
```

Listing 3.16: Graphology nodes with Sigma.js attributes

An event handler can be added to the rendered graph, like in Listing 3.17.

```
renderer.on('enterNode', ({ node }) => {
  // print node entered with the mouse cursor
  console.log(node);
});
```

Listing 3.17: Sigma.js event handler example

The event handler is a function executed when a user performs a certain interaction with the rendered graph. Sigma.js provides multiple types of event handlers. The example above includes the `enterNode` event handler, which occurs when a node is entered with the mouse cursor. Other events include leaving the node with the cursor, clicking the node, or entering an edge with the cursor. Event handlers can be used to manage the state of the graph based on user interactions. For example, the mentioned `enterNode` event handler could be utilized to highlight hovered nodes.

Chapter 4

Product-related pages and user requirements

This chapter presents the state of the original UI on Product-related pages. Product-related pages contain information about Products and entities that are related to Products, i.e. Product Versions, Product Milestones, Builds, and Artifacts. This Bachelor's thesis aims to improve the visualization of mentioned Product-related entities and to portray relationships between them and their statistics; therefore, this chapter illustrates how this is handled on the original UI. Then, a description of user requirements regarding the improvement of those pages follows. The user requirements include issues PNC users complained about or suggestions made by PNC developers.

4.1 Product-related pages on the PNC build system original UI

This section describes the current state of Product-related pages on the original UI.

4.1.1 Artifacts list

Artifacts list is a table listing Artifacts. The table can contain all Artifacts in the PNC build system or a smaller set, such as Delivered Artifacts of a Product Milestone. Figure 4.1 shows the look of the list on the original UI.

The first column of the table does not parse the Artifact identifier in order to make it easy for the user to distinguish individual parts of it, such as the name and version of NPM Artifact, but rather the whole identifier is displayed as a plain string. Because of this, namely, Maven Artifact identifiers are hard to read.

A large part of the table is taken up by a column of checksums, even though it is not the most important column from the user's point of view, and checksums typically do not consume most of this column's width.

To see a Build that produced an Artifact, the user first needs to open the Artifact detail page (the link to it is in the list) where the link to the Build is contained. The link to the Build is not included in the list itself.

The list is paginated (divided into multiple pages) and has filtering and sorting options. The issue with all lists on the original UI is that the current page displayed by the list or the

configured filtering and sorting options of the list are not persisted in the URL. Therefore, the URL of the list cannot be bookmarked or sent to someone else to display the same data.

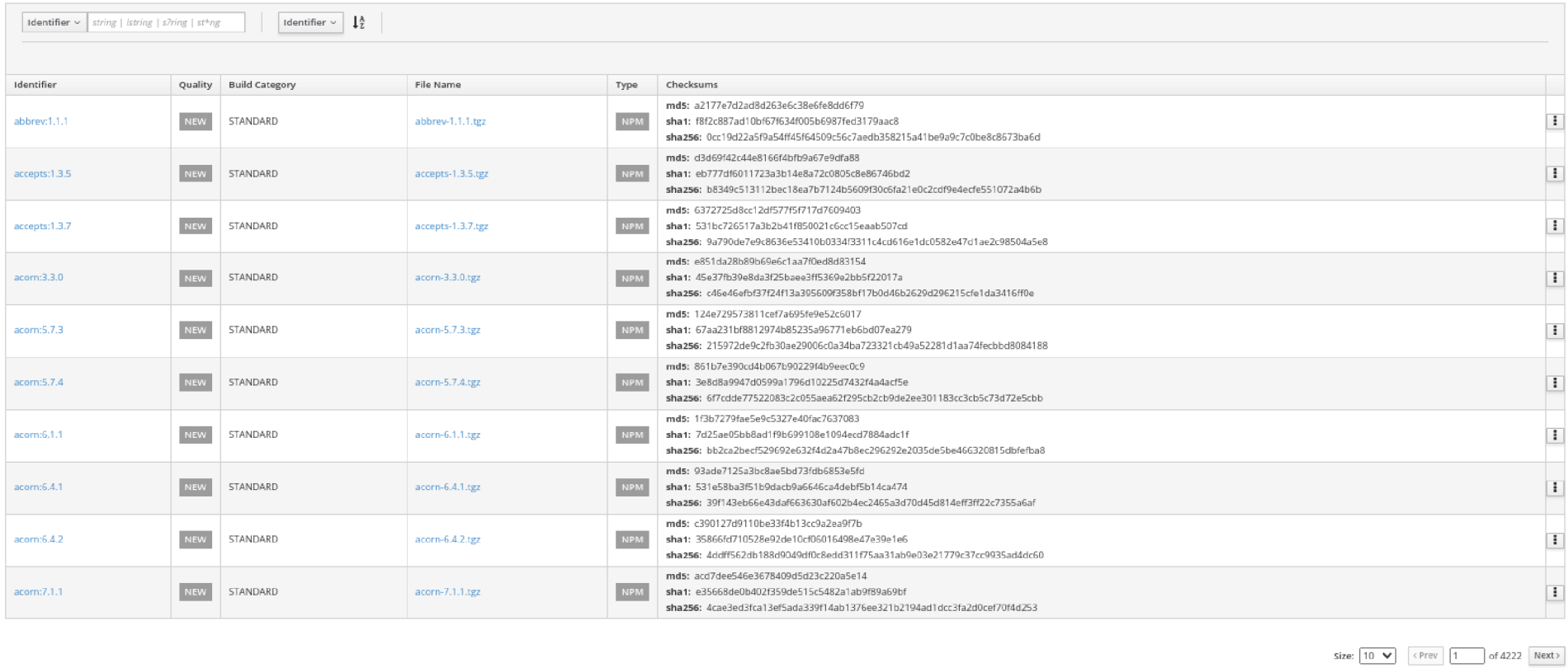

Figure 4.1: Artifacts list on the PNC build system original UI

4.1.2 Product Milestone pages

Specific Product Milestone has a set of its own pages divided into tabs. They contain Product Milestone's properties and lists of entities belonging to a Product Milestone. Figure 4.2 shows one of Product Milestone's pages, the detail page, on the original UI. The detail page's content is mostly empty. It includes only the most important properties of a Product Milestone, so-called Product Milestone details. All other data related to the Product Milestone are on their separate tabs, such as the Delivered Artifacts list, for example.

The Delivered Artifacts list provides the Delivered Artifacts of one Product Milestone. The original UI does not provide a way to compare Delivered Artifacts of two or multiple Product Milestones or to display a list of shared Delivered Artifacts between Product Milestones. To do that, it is required to go through Product Milestones of interest and their Delivered Artifacts lists manually.

Product Milestone pages also offers no summary of data about one Product Milestone and its related entities, for example, Builds and Produced or Delivered Artifacts, in the form of statistics and charts of their counts and properties. To get these data about related entities on the original UI it is necessary to analyze these entities manually. Therefore, open their detail pages and extract the information individually. For example, to see the source of Delivered Artifacts of a Product Milestones, the user must open their detail pages and find which Build they were produced by (if any) and then open Build detail pages to find out what Product Milestone that Builds belong to (if any). And to find out the statistics of Produced Artifacts, the user has to go through all Builds belonging to a Product Milestone and there through all their Artifacts produced by the Build.

Figure 4.2: Product Milestone detail page on the PNC build system original UI

4.1.3 Product Version page

Unlike Product Milestone, the specific Product Version is not divided into multiple pages on their own tabs, as is illustrated in Figure 4.3. Rather, there is only one detail page containing important properties (the so-called Product Version details) in the key-pair list along with tables of entities belonging to the Product Version.

The Product Version page faces the same problem as the Product Milestone pages. There is no direct way to get the statistics about one Product Version or the summary of Produced or Delivered Artifacts of all Product Milestones of the Product Version. On the Product Version level, the user would need to collect the data manually, and it is more inefficient since it needs to be done for all Product Milestones of a Product Version.

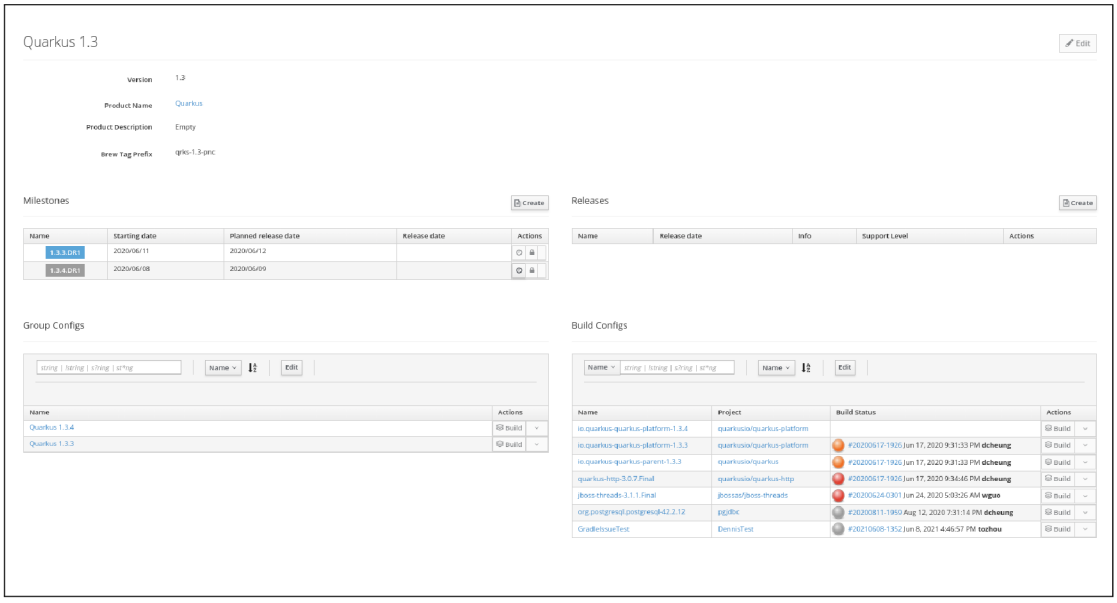

Figure 4.3: Product Version detail page on the PNC build system original UI

4.2 User requirements

This section lists user requirements regarding the improvement of the Product-related pages. The requirements were either suggested by PNC users or by developers.

4.2.1 Artifacts list

There are three requirements for the new Artifacts list that would improve its usage.

- an option to dissect the Artifact identifier and its individual parts and separate the parts visually – that way, the user can easily see specific parts of the identifier, such as the Artifact archive type or Artifact version
- a column of checksums needs to be shrunk; checksums could be hidden in an expandable row, too
- a link to the Build that produced an Artifact in the list
- pagination, sorting, and filtering options of the list need to be persisted in the URL

4.2.2 Product Milestone pages

The requirement for the Product Milestone page is to convert the Product Milestone detail page into a dashboard. The dashboard needs to contain statistics and charts summarizing information about a Product Milestone and entities belonging to it.

Information users find useful is related to Used, Produced, and Delivered Artifacts of a Product Milestone. More specifically summary of their source and distribution of their properties. The data can be visualized in the form of data cards and charts. Charts need to be accompanied by legends.

The source of Artifacts can be visualized as the number of Artifacts coming from specific sources, for example, the number of Artifacts produced by a Build in a Milestone or the number of Artifacts not produced by any Build.

The distribution of properties of Artifacts includes the distribution of Artifact quality or the distribution of repository type. With those distributions, users would be able to see, for example, whether there is a higher amount of Artifacts with problematic quality (such as `deprecated`). Also, for quick reference, the distribution of repository type would help the user to learn from which type of repository (such as Maven or NPM) Artifacts came.

Different users use different screen sizes; therefore, the dashboard should be responsive to at least a certain degree so that the charts and statistics are responsively resized. Additionally, the dashboard layout should change responsively to better fit certain screens.

4.2.3 Product Version pages

To keep the page structure consistent with the Product Milestone pages, Product Version details need to be separated from the tables of related entities. Tables of entities belonging to the Product Version would have separate pages.

In place of tables, new statistics and charts would come to build a new dashboard. The Product Version dashboard should mainly aggregate data from Product Milestones of the Product Version. For example, in the distribution of the source of Delivered Artifacts, Delivered Artifacts in all Product Milestones of the Product Version would be aggregated.

4.2.4 Network graphs

Some users asked for a new feature of network graphs that visualizes relationships between PNC entities. Nodes would represent entities and edges relationships between them. These include the following:

- Sharing of Used or Delivered Artifacts between Product Milestones
- Sharing of Used or Delivered Artifacts between Product Versions
- Build-time dependencies between Builds
- Build-time dependencies between Product Milestones

In the graph, the user should be able to open either the list of shared Artifacts between entities or a list of dependencies forming the dependency of one entity on another.

Should the graph contain hundreds of entities and relationships between those entities, creating a big web of nodes in the graph, there needs to be a way to highlight certain data and de-emphasize other. Therefore, the requirement is to provide ways to highlight or hide nodes in order to make the graph more readable. Furthermore, filtering of nodes would be beneficial.

4.2.5 Product Milestone comparison

Product Milestone Comparison is a UI component directly suggested by a PNC user. The component is a table allowing users to compare Delivered Artifacts of two or more selected Product Milestones and their versions. The table would list Delivered Artifacts shared between at least two of the selected Product Milestones.

The first column would display the identifiers of the Delivered Artifacts (without the Artifact version). Other columns would represent the selected Product Milestones. These columns would list versions of the Delivered Artifacts in the corresponding Product Milestones. Links to the pages of the Delivered Artifact versions should be provided, along with links to the Builds that produced them.

Chapter 5

UI design

This chapter presents the new Product-related pages or the original ones which were re-designed. Their design adheres to the user requirements specified in Chapter 4. These designs are implemented on the new UI.

Wireframes[1] are provided to illustrate the design along with the description. Wireframes were created in the Figma[2] web application.

5.1 Artifacts list

The new design of the Artifacts list is included in Figure 5.1.

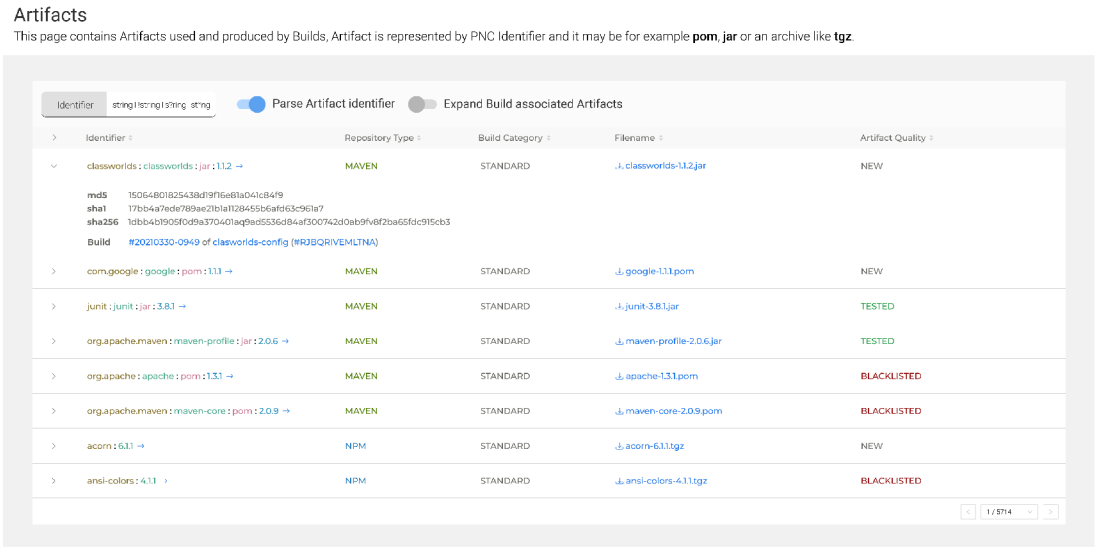

Figure 5.1: Artifacts list wireframe

Each row in a list represents an Artifact. The rows in the table are expandable and row expansion is controlled by the button on the left side of the row. The expandable row area contains all checksums including a brand new link to a Build which produced an Artifact. The “expand all” button in the top-left corner of the table expands all rows of the table at

[1] https://en.wikipedia.org/wiki/Website_wireframe
[2] https://figma.com/

once. There is also a toggle element to expand all Artifact rows which contain a link to the Build, therefore it expands Artifacts produced in the PNC build system.

Another toggle in the table makes Artifact identifiers parsed and their individual parts distinguished by a different color. When Artifact identifier parsing is on, a link to the Artifact detail is included in the form of a button located to the right of the identifier. When the feature is off, the identifier string itself represents the link.

Some values in the table rows are differentiated by distinct colors, such as repository type or Artifact quality. For example, `blacklisted` Artifact quality is highlighted in red color.

5.2 Product Milestone dashboard

The wireframe in Figure 5.2 depicts the design of the Product Milestone dashboard.

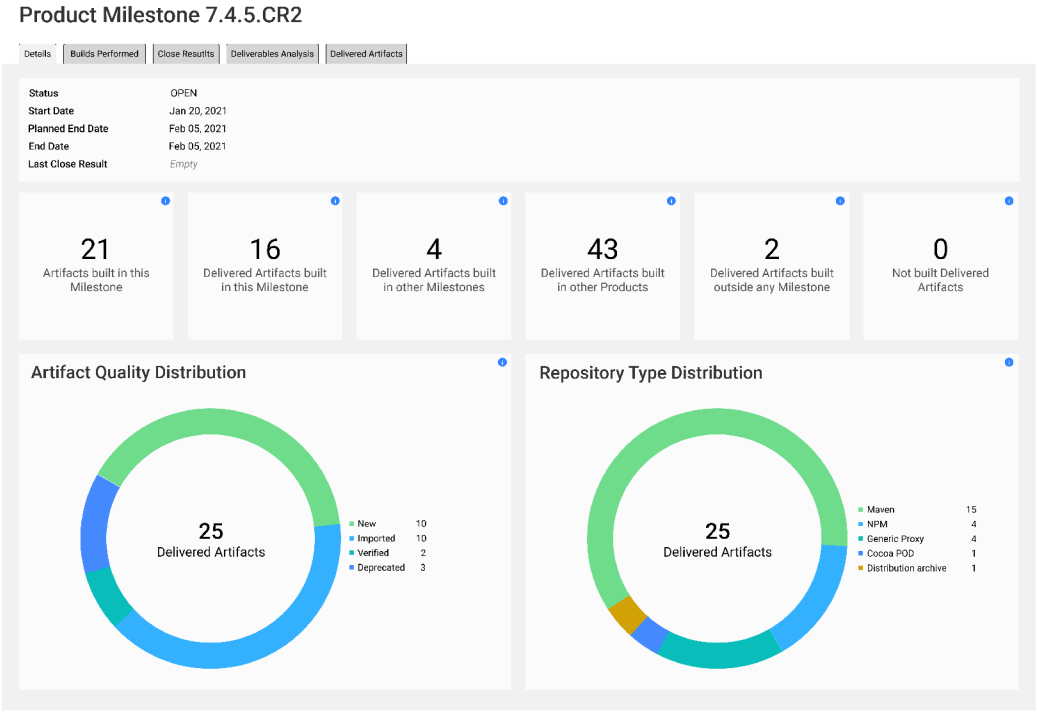

Figure 5.2: Product Milestone dashboard wireframe

The Product Milestone dashboard extends the Product Milestone detail page to fill in empty spaces and provide statistics about a Product Milestone. The entire dashboard is divided into visually separated boxes. The page begins with a box of the original Product Milestone details. The new dashboard's content is placed under the details. First, there is a set of data cards of various statistics about Produced and Delivered Artifacts of a Product Milestone. Then two charts of the distribution of Delivered Artifacts follow. Data cards and chart boxes have a tooltip in the top right corner to provide additional explanations on the meaning of the data they represent.

The data cards hold data about the Produced Artifacts and the source of Delivered Artifacts. Section 2.3.1 provides information about the source of Artifacts. In the end, the following statistics were chosen.

- **first card** – the number of Artifacts produced by Builds contained in the Product Milestone of the detail page
- **second card** – the number of Delivered Artifacts produced by Builds contained in the Product Milestone of the detail page
- **third card** – the number of Delivered Artifacts produced by Builds contained in Product Milestones which belong to the same Product as the Product Milestone of the detail page
- **fourth card** – the number of Delivered Artifacts produced by Builds contained in Product Milestones which belong to different Products than the Product Milestone of the detail page
- **fifth card** – the number of Delivered Artifacts produced by Builds not contained in any Product Milestone
- **sixth card** – the number of the Delivered Artifacts that were not built at all in the PNC build system

On the card, the number is highlighted by a larger font, and the card description is de-emphasized by gray color. This is because data are the most important information and descriptions are secondary [12, pp. 50-51].

Both charts display the distribution of Delivered Artifacts. The first chart shows the distribution of Artifact quality of Delivered Artifacts. The second one represents the distribution of repository type of Delivered Artifacts.

5.3 Product Version dashboard

The wireframe for the Product Version Dashboard is included in Figure 5.3. Delivered Artifacts set in this section refers to all Delivered Artifacts of all Product Milestones of the Product Version to which the dashboard corresponds.

The new Product Version detail page design separates the original detail page into multiple pages. This way, the page structure is consistent with the Product Milestone variant. Overall, the Product Version detail page/dashboard follows the same design as the Product Milestone one. The Product Version dashboard extends the detail page, so the first box holds the Product Version details. Some of the data cards are aggregated forms of their counterparts on the dashboards of Product Milestones of the Product Version the detail page belongs to.

The following values were chosen for the data cards.

- **first card** – the number of Products that contain dependency Milestones; dependency Milestone contains Build or Builds that produced some of the Delivered Artifacts
- **second card** – the number of dependency Milestones
- **third card** – the number of Artifacts produced by Builds contained in Product Milestones of the Product Version of the detail page
- **fourth card** – the number of Delivered Artifacts produced by Builds contained in Product Milestones of the Product Version of the detail page

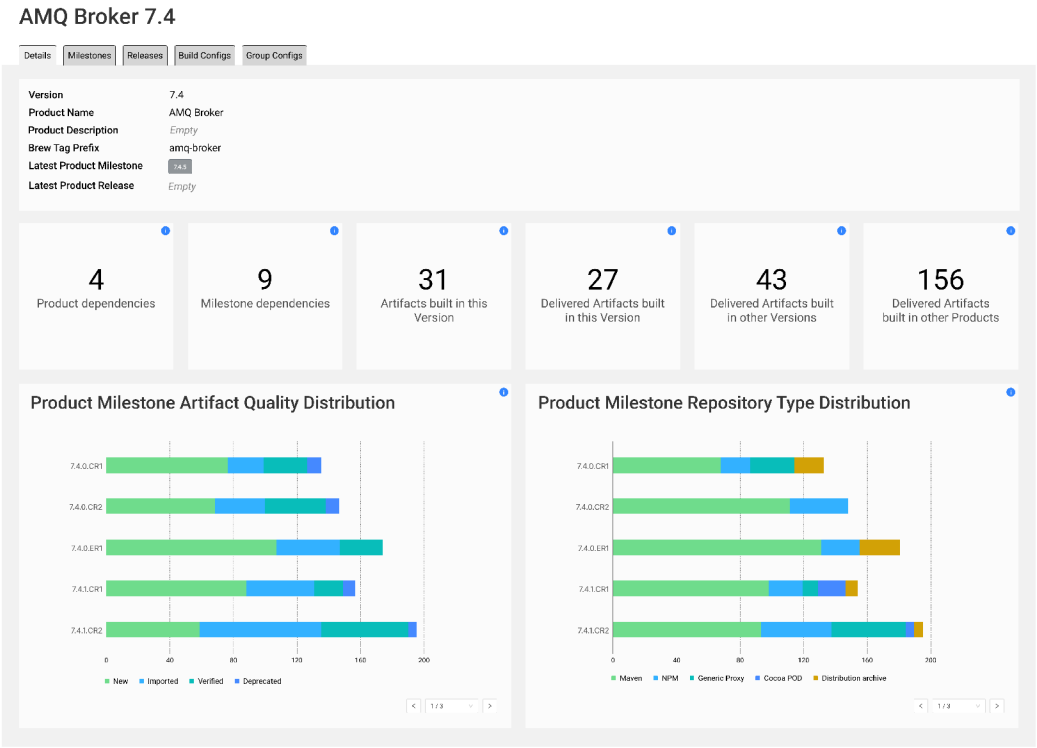

Figure 5.3: Product Version dashboard wireframe

- **fifth card** – the number of Delivered Artifacts produced by Builds contained in Product Milestones of other Product Versions than the Product Version of the detail page, but contained in the same Product
- **sixth card** – the number of Delivered Artifacts produced by Builds contained in Product Milestones of different Products than the one the Product Version of the detail page belongs to

Charts aggregate their counterparts on dashboards of Product Milestones of the Product Version of the detail page.

5.4 Product Milestone interconnection graph

Out of various graphs suggested in the discussion with users and developers (listed in Section 4.2.4), two were designed to be implemented, the first being the Product Milestone interconnection graph. The wireframe for it is displayed in Figure 5.4. The design of this graph will serve as an example model of the network graph, and other graphs will follow it. Section 2.3.1 explains relationships between Product Milestones.

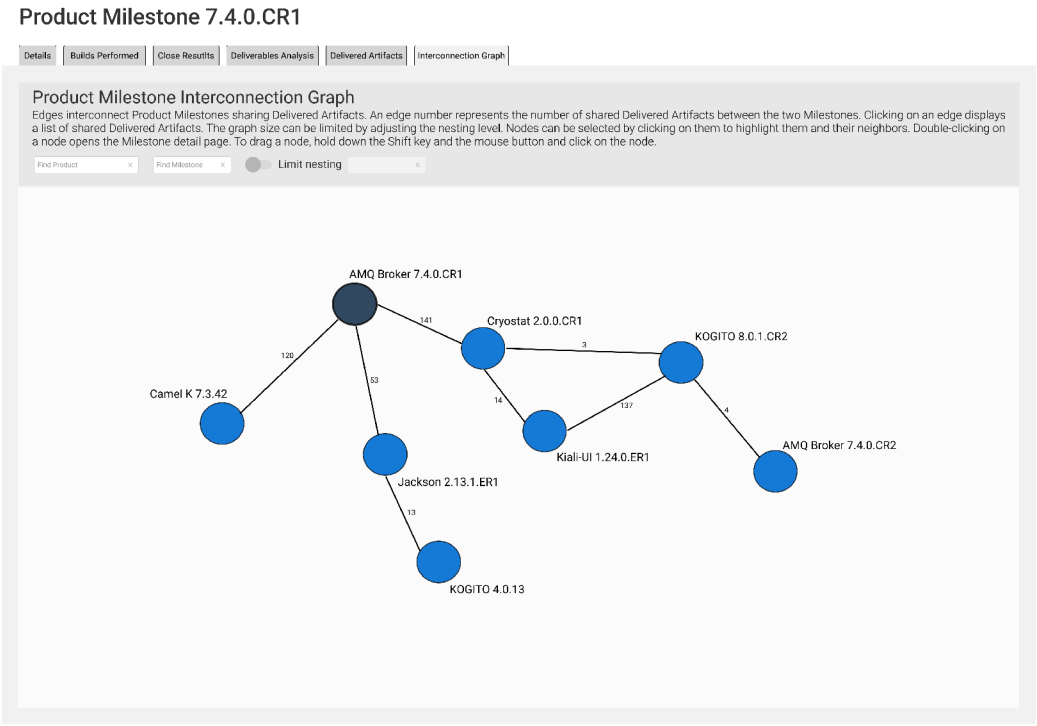

Figure 5.4: Product Milestone interconnection graph wireframe

The Product Milestone interconnection graph was devised to be put on its separate page in the Product Milestone pages structure since the graph belongs to a single Product Milestone. The Product Milestone to which the graph page corresponds will be referred to as the main Product Milestone.

The network graph visualizes Product Milestones sharing Delivered Artifacts with the main Product Milestone. The same sharing of the Delivered Artifacts is visualized also for Product Milestones the main Product Milestone shares Delivered Artifacts with. It needs to be noted that there is no transitive relationship between these interconnections in the graph – if Product Milestone A shares Delivered Artifacts with Product Milestone B and Product Milestone B shares Delivered Artifacts with Product Milestone C, it does not implicate sharing of Delivered Artifacts between Product Milestones A and C.

Each Product Milestone is represented as one graph node. The node displays the name of the Product Milestone along with the name of the Product to which the Product Milestone belongs. The main Product Milestone is highlighted in the graph (by a darker color in the wireframe). Edges interconnecting Product Milestones sharing Delivered Artifacts have labels on them, providing the count of shared Delivered Artifacts.

The wireframe portrayed in Figure 5.5 illustrates a feature that allows limiting the nesting of the graph from the main node.

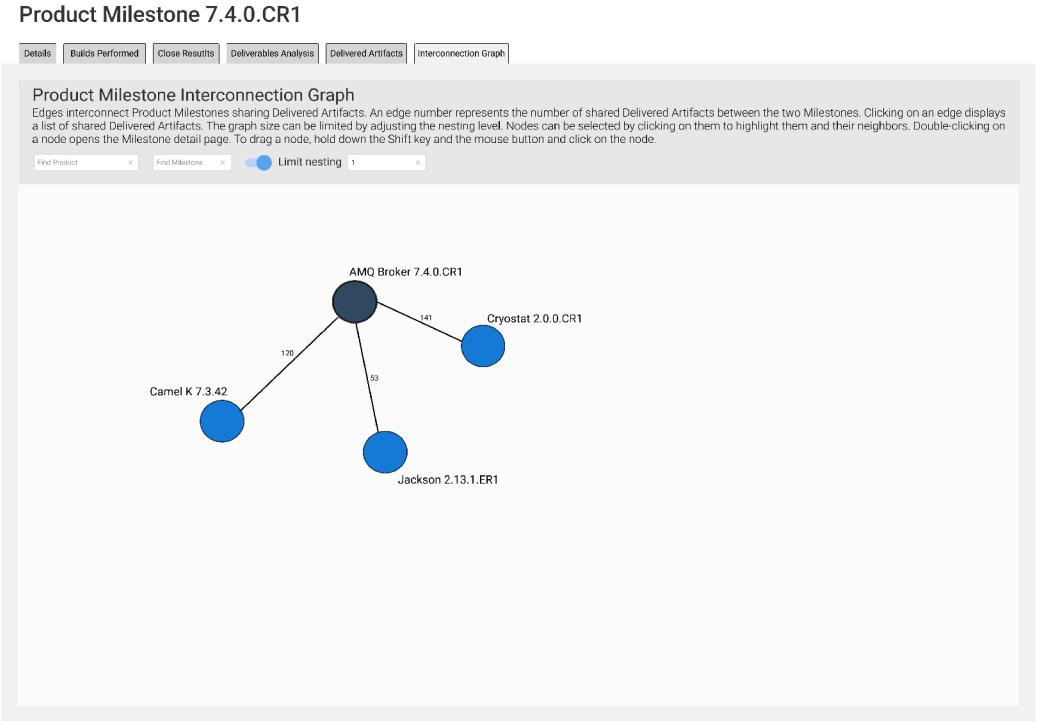

Figure 5.5: Product Milestone interconnection graph wireframe – nesting level limitation

This feature restricts to what level from the main Product Milestone interconnections of the Product Milestones are nested. For example, the nesting level with value 1 shows only interconnections of the main Product Milestone, and the nesting level with value 2 also interconnections of those Product Milestones which share Delivered Artifacts with the main Product Milestone. This feature is useful especially when the user is interested to see only which Product Milestones share Delivered Artifacts with the main Product Milestone (for which scenario nesting level 1 would be set). Another use case is to make the graph more readable should it contain a large number of nodes.

To satisfy the need to make the graph cleaner in situations when the graph contains a great number of nodes, other features were designed.

The wireframe in Figure 5.6 presents the ability of the user to hover over a graph node to highlight the node including its neighbors. The other nodes in the graph are grayed out. A similar feature is node selection. By clicking on the node, the node is marked as selected. Selected nodes are displayed as hovered nodes, that is, the nodes and their neighbors are highlighted, and other nodes are grayed out. This allows the user to explore the graph and de-emphasize the data the user finds less interesting.

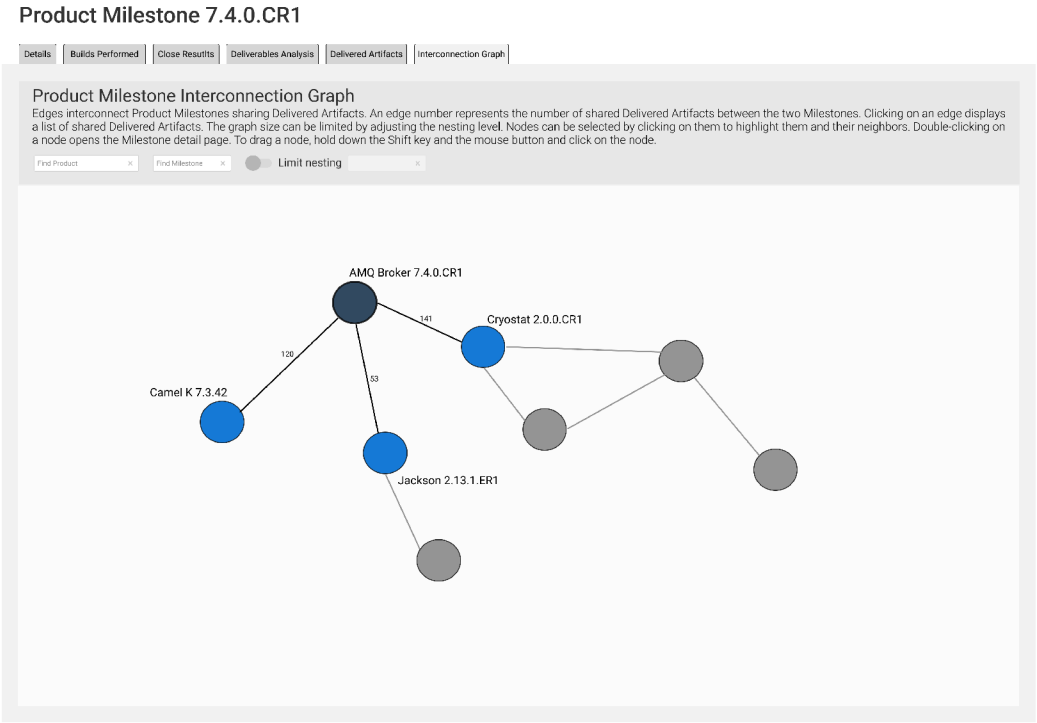

Figure 5.6: Product Milestone interconnection graph wireframe – hovering over the node

The graph page also has two search bars. The first search bar looks for Product Milestones belonging to a certain Product. The second search bar matches Product Milestone's name. The matched Product Milestones are highlighted by a special color. If both search bars are used, the intersection of matched Product Milestones is highlighted.

Another important feature is a list of Delivered Artifacts shared between two Product Milestones, displayed on the wireframe in Figure 5.7. The list is displayed by clicking on any edge connecting two Product Milestones and contains Delivered Artifacts shared between these two Product Milestones. The list, along with an identifier of the Delivered Artifact, includes a link to a Build that produced the Delivered Artifact (source Build), a link to a Product Milestone in which that Build is contained (source Product Milestone), and a link to a Product to which that Product Milestone belongs (source Product).

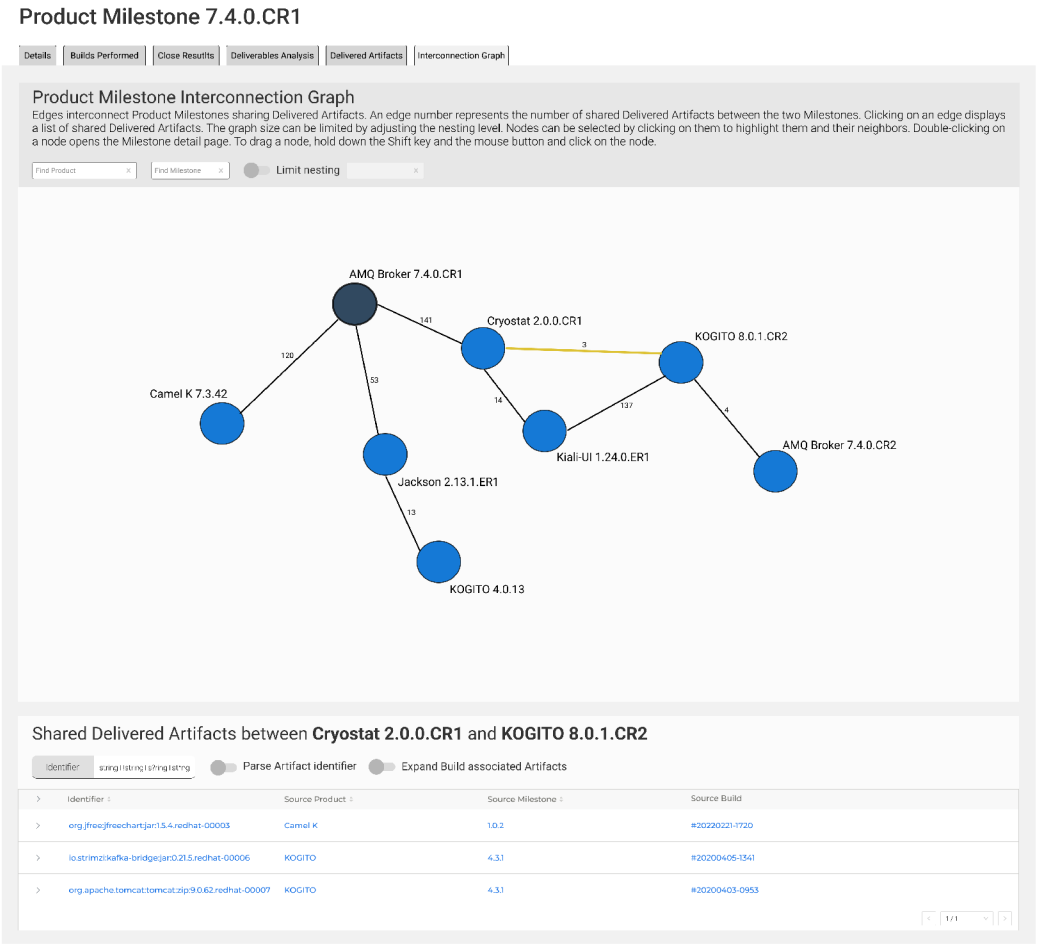

Figure 5.7: Product Milestone interconnection graph wireframe – shared Delivered Artifacts list

5.5 Build Artifact dependency graph

The second of two designed network graphs is the Build Artifact dependency graph of build-time dependencies. Its wireframe is shown in Figure 5.8. Section 2.4 describes the build-time dependencies.

Nodes represent the Builds. This graph is directed, so Builds point to other Builds. The edge arrow points from dependent Build to Build it depends on. The edge label displays the count of Artifacts dependent Build uses from Build it depends on.

The design of features follows the same pattern as the Product Milestone interconnection graph. It has the same feature of node highlighting by hovering over nodes or selecting nodes. Limitation of nesting level is also present. Users can search for Build by its name or the name of the Build Configuration. By clicking on the edge, the list of Used Artifacts is shown.

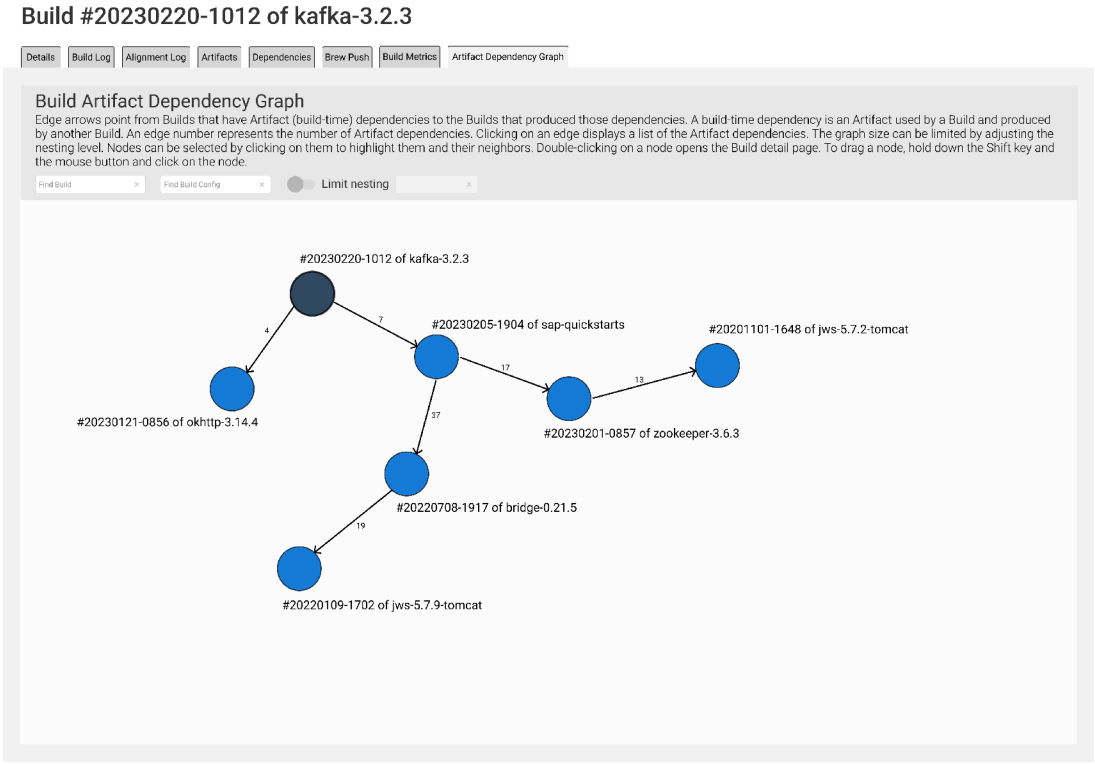

Figure 5.8: Build Artifact dependency graph wireframe

5.6 Product Milestone comparison

The Product Milestone comparison is depicted in the wireframe in Figure 5.9. The table compares the versions of Delivered Artifacts of Product Milestone the user selects.

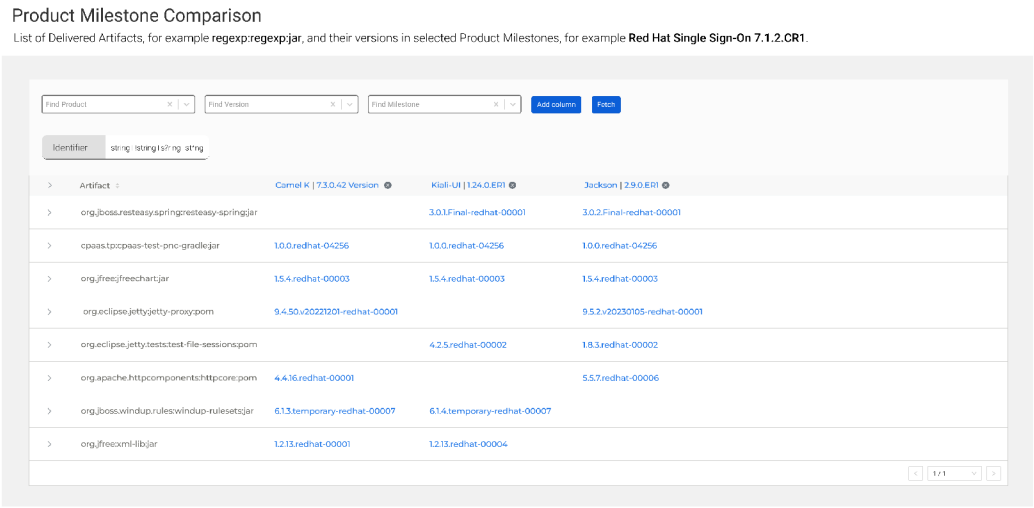

Figure 5.9: Product Milestone comparison table

Firstly, the user needs to select Product Milestones to be compared. By using three search bars for Product, Product Version, and Product Milestone, respectively, Product Milestones of interest are found. Found Product Milestone can be added to the table header by the "Add column" button. Then, when the Product Milestones were selected, the "Fetch" button can be used to fetch the table data.

An alternative to this approach would be to fetch the table data each time the Product Milestone is selected in the search bar, and no fetch button would be necessary. But this would lead to needless requests if more than two Product Milestones were selected, so it was avoided.

Product Milestones can be also deselected from the table, or new Product Milestones can be added to the table, and then the table data can be fetched again.

Chapter 6

REST API design

This chapter describes the REST API[1] designed for endpoints needed by the pages designed in Chapter 5 including the chosen HTTP method and their data format. An important note is that this Bachelor's thesis does not implement the final backend for these endpoints, just their design is laid out. TypeScript syntax for interfaces describes the data format of the endpoints.

6.1 Pagination in the PNC REST API

PNC REST API uses the following structure for the pagination of mostly tabular data. This data structure (`Page`) is referred to later in this chapter when the pagination of some endpoint data is needed.

```
interface Page {
    content?: any[];
    totalPages?: number;
    pageIndex?: number;
    pageSize?: number;
    totalHits?: number;
}
```

Properties of the page of data:

- **totalPages** – the count of pages data are divided into
- **pageIndex** – the index of one page currently returned by the backend
- **pageSize** – the number of data entities that are maximally returned in one page
- **totalHits** – the total count of all existing data entities for a specific endpoint
- **content** – an array of data entities corresponding to the endpoint; for example, for the Artifacts list endpoint, it is an array of Artifacts

[1] https://www.redhat.com/en/topics/api/what-is-a-rest-api

6.2 Product Milestone dashboard

The existing GET endpoint `/product-milestones/:id` gets the details of a Product Milestone on the Product Milestone dashboard. New dashboard statistics, including data cards and charts, were designed to have their own endpoint rather than extending the existing detail endpoint with new data.

GET /product-milestones/:id/statistics

This endpoint returns data card statistics and chart data.
Returned data format (`ProductMilestoneStatistics`):

```
interface ProductMilestoneStatistics {
    artifactSource: {
        // the number of Artifacts produced by Builds
        // contained in the requested Milestone
        thisMilestone: number;
    }
    deliveredArtifactsSource: {
        // the number of the Delivered Artifacts produced by Builds
        // contained in the requested Milestone
        thisMilestone: number;
        // the number of the Delivered Artifacts produced by Builds
        // contained in Milestones which belong to the same Product
        // as the requested Milestone
        previousMilestones: number;
        // the number of the Delivered Artifacts produced by Builds
        // contained in Milestones of other Products
        otherProducts: number;
        // the number of the Delivered Artifacts produced by Builds
        // not  contained in any Milestone
        noMilestone: number;
        // the number of the Delivered Artifacts not produced by any Build
        noBuild: number;
    }
    // proportion of quality of Delivered Artifacts
    artifactQuality: {
        [key: string]: number;
    }
    // proportion of repository type of Delivered Artifacts
    repositoryType: {
        [key: string]: number;
    }
}
```

The whole data structure is nested and the properties are grouped logically. Artifact quality and repository type properties are objects whose keys are individual Artifact qualities or repository types and values are numbers of Delivered Artifacts corresponding to the Artifact quality or repository type.

6.3 Product Version dashboard

On the Product Version dashboard, three new endpoints for dashboard statistics were designed rather than extending existing detail GET endpoint `/product-versions/:id`.

GET /product-versions/:id/statistics

This endpoint returns data card statistics (not chart data).
Returned data format (`ProductVersionStatistics`):

```
interface ProductVersionStatistics {
    // the number of Milestones created in the requested Version
    milestones: number;
    // the number of Products which contain Milestones
    // containing Builds which produced the Delivered Artifacts
    productDependencies: number;
    // the number of Milestones which contain Builds which produced
    // the Delivered Artifacts
    milestoneDependencies: number;
    artifactSource: {
        // the number of Artifacts produced by Builds contained
        // in Milestones of the requested Version
        thisVersion: number;
    }
    deliveredArtifactsSource: {
        // the number of the Delivered Artifacts produced by Builds
        // contained in Milestone of the requested Version
        thisVersion: number;
        // the number of the Delivered Artifacts produced by Builds
        // contained in Milestones of other Versions of the same Product
        previousVersions: number;
        // the number of the Delivered Artifacts produced by Builds
        // contained in Milestones of other Products
        otherProducts: number;
        // the number of the Delivered Artifacts produced by Builds
        // not contained in any Milestone
        noMilestone: number;
        // the number of the Delivered Artifacts not produced by any Build
        noBuild: number;
    }
}
```

Once again, the structure is logically nested. Not all of the data provided by this endpoint are used in the final dashboard design but providing them allows UI to be easily expandable for some new information if such need arises. Artifact quality and repository type distribution charts were separated into their own endpoints.

GET /product-versions/:id/artifact-quality-statistics

This endpoint returns Artifact quality distribution chart data.
Returned data format (`ProductVersionArtifactQualityStatisticsPage`):

```
interface ProductVersionArtifactQualityStatisticsPage extends Page {
    content?: ProductVersionArtifactQualityStatistics[];
}

interface ProductVersionArtifactQualityStatistics
extends ProductMilestoneRef {
    artifactQuality: {
        [key: string]: number;
    }
}
```

The endpoint returns an array of objects containing Artifact quality distribution for Product Milestones of a Product Version. Also, the shorter version of the Product Milestone detail is included (`ProductMilestoneRef` extending chart data). This endpoint was separated from the data card statistics endpoint due to its pagination. If it was merged with the that endpoint, the same data card statistics would be returned each time a new page is requested. The reason for the pagination of this endpoint is that Product Version can contain an unlimited amount of Product Milestones, therefore unpaginated endpoint could return a large batch of data. The other chart needs to be paginated too, and both charts are paginated separately, so the repository type distribution chart has its own endpoint too.

GET /product-versions/:id/repository-type-statistics

This endpoint returns repository type distribution chart data.
Returned data format (`ProductVersionRepositoryTypeStatisticsPage`):

```
interface ProductVersionRepositoryTypeStatisticsPage extends Page {
    content?: ProductVersionRepositoryTypeStatistics[];
}

interface ProductVersionRepositoryTypeStatistics
extends ProductMilestoneRef {
    repositoryType: {
        [key: string]: number;
    }
}
```

This endpoint is separated from the data card statistics endpoint for the same reason as the Artifact quality distribution chart endpoint. The data structure is the same, except, instead of Artifact quality distribution, repository type distribution is included for Product Milestones of a Product Version.

6.4 Product Milestone interconnection graph

For the Product Milestone interconnection graph, one endpoint was designed for returning the interconnection graph of a Product Milestone and one endpoint for returning shared Delivered Artifacts between two Product Milestones.

GET /product-milestones/:id/interconnection-graph

This endpoint returns the graph.
Returned data format (`ProductMilestoneInterconnectionGraph`):

```
interface ProductMilestoneInterconnectionGraph {
    vertices: {
        [key: string]: {
          // Milestone ID
          name: string;
          // Milestone data
          data: ProductMilestone;
        };
    }
    edges: ProductMilestoneInterconnectionGraphEdge[];
}

interface ProductMilestoneInterconnectionGraphEdge {
    source: string;
    target: string;
    // number of Delivered Artifacts shared between Milestones of the edge
    sharedDeliveredArtifacts: number;
}
```

The data structure is divided into vertices and edges. This fits the way nodes and edges are added to the graph in Sigma.js – they are added separately. `vertices` is an object whose keys are IDs of Product Milestones included in the graph and values are data of those Product Milestones. Theoretically, this could be an array of Product Milestones, but the object allows a specific Product Milestone to be found by its ID directly. The edge has its source node and its target node. `source` and `target` contain IDs of the Product Milestones that the edge interconnects.

GET /product-milestone-shared-delivered-artifacts

This endpoint returns the list of shared Delivered Artifacts. The endpoint accepts two query parameters to determine the Product Milestones for which the list should be fetched – `milestone1` and `milestone2`, each containing ID of the Product Milestone.
Returned data format (`ProductMilestoneSharedDeliveredArtifactsPage`):

```
interface ProductMilestoneSharedDeliveredArtifactsPage extends Page {
    content?: ProductMilestoneSharedDeliveredArtifact[];
}

interface ProductMilestoneSharedDeliveredArtifact
```

```
extends Artifact {
    product: ProductRef;
    productVersion: ProductVersionRef;
    productMilestone: ProductMilestoneRef;
}
```

The endpoint returns a list of Delivered Artifacts. Delivered Artifact is extended by short detail of a Product (`ProductRef`), Product Version (`ProductVersionRef`), and a Product Milestone (`ProductMilestoneRef`) to which the Delivered Artifact belongs.

6.5 Build Artifact dependency graph

Build Artifact dependency graph needs two new endpoints, one for the graph itself and one for the list of Artifact dependencies.

GET /builds/:id/artifact-dependency-graph

This endpoint returns the graph.
Returned data format (`BuildArtifactDependencyGraph`):

```
interface BuildArtifactDependencyGraph {
    vertices: {
        [key: string]: {
          // Build ID
          name: string;
          // Build data
          data: Build;
        };
    }
    edges: BuildArtifactDependencyGraphEdge[];
}

interface BuildArtifactDependencyGraphEdge {
    source: string;
    target: string;
    // number of Artifacts source Build depends on
    artifactDependencies: number;
}
```

This graph follows the same data pattern as the Product Milestone interconnection graph. The keys of `vertices` object are IDs of Builds and values are data of the Builds. `source` and `target` of the edge are IDs of the Builds.

GET /build-artifact-dependencies

This endpoint returns the list of Artifact dependencies of one Build depending on another. The endpoint accepts two query parameters – `dependentBuild` and `dependencyBuild` which determine edge of the graph.
Returned data format (`BuildArtifactDependenciesPage`):

```
interface BuildArtifactDependenciesPage extends Page {
    content?: Artifact[];
}
```

6.6 Product Milestone comparison

For the Product Milestone comparison table, one POST endpoint was designed.

POST /product-milestone-comparison

This endpoint returns the compared Delivered Artifacts in selected Product Milestones. Input data format (`ProductMilestoneComparisonInputData`):

```
interface ProductMilestoneComparisonInputData {
    productMilestones: string[];
}
```

Product Milestones to be compared are sent as a list of their IDs. Due to this fact, the POST HTTP method was selected for the endpoint.
Returned data format (`ProductMilestoneComparisonPage`):

```
interface ProductMilestoneComparisonPage extends Page {
    content?: ComparedArtifact[];
}

interface ComparedArtifact {
    // Artifact identifier without the version
    identifier: string;
    productMilestones: {
        [key: string]: ArtifactWithVersion;
    }
}

interface ArtifactWithVersion extends Artifact {
  // version of the Artifact
  artifactVersion: string;
}
```

The endpoint returns an array of Delivered Artifacts and their versions in selected Product Milestones. `productMilestones` object's keys are IDs of the Product Milestones and values are concrete Delivered Artifacts of the corresponding Product Milestone extended by the name of their version (`artifactVersion`).

Chapter 7

Implementation of the designed pages

This chapter explains the implementation of the pages designed in Chapter 5. The first section lists the structure of the code. The second section describes HTTP services used by the new UI project. The third section provides information about components implemented before this Bachelor's thesis. The first three sections summarize the state before the implementation part of this Bachelor's thesis started. The fourth section of this chapter delves into the implementation part by describing a chosen set of implemented React components and functions. The last section outlines the testing of the implemented features.

7.1 Code structure

The new PNC UI project is developed in an open-source GitHub repository [10]. The code is structured in the following directory tree:

- `.github/`
  - `workflows/` – Github Workflows
- `documentation/` – Documentation and READMEs
- `src/` – Application source code
  - `common/` – Code and constants used in the whole project
  - `components/` – React components
  - `hooks/` – Custom React hooks
  - `libs/` – Code to use with third-party libraries
  - `services/` – HTTP services
  - `utils/` – Various helper functions, data transformations, etc.

7.2 HTTP services

The new UI project implements a connection to the REST API of the HTTP servers through Axios instances. These hold base URLs to which all HTTP requests will be sent and provide

methods to send specific HTTP requests, such as the `get` method. Axios instances are then encapsulated in singleton objects. An example is `pncClient` creating an interface for the PNC REST API.

Some Axios instances also have interceptors preprocessing HTTP requests. The interceptor in `pncClient`, for example, adds authentication headers identifying the PNC user to the HTTP request or sets the correct content type of HTTP PATCH and POST requests.

Service functions (or services) then implement connections to specific REST endpoints. All services create HTTP requests with the help of Axios instances inside client singleton objects. Service fetching the Product Milestone interconnection graph is included in Listing 7.1.

```
export const getInterconnectionGraph = ({ id }, requestConfig) => {
  return mockClient
            .getHttpClient() // returns Axios instance
            .get(`/product-milestones/${id}/interconnection-graph`,
                 requestConfig);
};
```

Listing 7.1: Product Milestone interconnection graph service in the new UI project

Since endpoints used by newly implemented components are not implemented as part of this Bachelor's thesis, their HTTP services use `mockClient` which sends all requests to the localhost port where mocked backend server should run.

7.2.1 Service containers

`useServiceContainer` is a custom hook that the new UI project uses to manage the execution, state, and data of HTTP services. The hook accepts a callback of a service as a parameter. It returns an object containing data that were returned by a backend (if any were loaded), an error (if any occurred), a boolean state as to whether the service is currently in a loading state, and a function to execute the service with (`run` function). When the `run` function is executed, it runs the input service callback to fetch the data. The object returned by the `useServiceContainer` hook is referred to as a service container.

A `ServiceContainerLoading` component receives the service container as its parameter and the child component instance. The child component is typically dependent on the data returned by the service. `ServiceContainerLoading` does not run the service but detects its state and displays adequate content. For example, if the service is in a loading state for the first time, the spinner is displayed. If the data were loaded, the child component is rendered. If the service is in a loading state and the data were loaded before, the child component is grayed out, and the loading bar is displayed on top of it.

7.3 Old components

This section briefly introduces components in the new PNC UI project that were not implemented as part of this Bachelor's thesis.

Pagination component

A `Pagination` component is used to display a pagination UI element allowing users to navigate the pages of the paginated endpoints. Paginated endpoints are those endpoints whose data are not returned in one batch but are divided into multiple pages. The component is mainly dedicated to tables. The component persists the pagination in the URL.

ContentBox component

A `ContentBox` component is a box encapsulating other components. It provides multiple parameters, such as displaying the white background and padding around the content of the box.

SearchSelect component

A `SearchSelect` component is a select whose options are dynamically fetched. It accepts a callback that fetches the options of the select. It also provides a search bar to narrow down the select options. Its UI is rendered by the `Select` PatternFly component.

7.4 Implemented components and functions

This section describes the implementation part of this Bachelor's thesis and the chosen set of components and functions.

Components whose names end with **Page** represent pages of the website. Each page has its own title and URL route, optionally a description.

7.4.1 Artifacts list

The Artifacts list page is created by an `ArtifactsPage` component.

ArtifactsList component

An `ArtifactsList` component's table is implemented using the `TableComposable` PatternFly component. The component is meant to be reusable on multiple pages; therefore, it is configurable and it is possible to choose the columns included in the displayed list. The component accepts a list of column identifiers to be displayed in the list, but the list has a default preset of shown columns. The component also accepts the service container of Artifacts to be listed in the table. The list conditionally renders the columns that are included in the input list of column identifiers.

Each Artifact row has its expandable row, where checksums and source Build are displayed. The component holds the state of currently expanded Artifact rows in an array of Artifact identifiers. When a button expanding row is clicked, the identifier is added to the array. Expanded rows are then conditionally rendered based on whether the identifier of an Artifact is included in that list. To ease up work with expandable rows, a button for expanding all rows and a toggle for expanding rows of Artifacts containing links to a Build are included in the list's toolbar. The button updates a state `areAllArtifactsExpanded` and the toggle updates a state `areBuildArtifactsExpanded`. `useEffect` hooks, containing those states and state of service container's data in a dependency array, are then used to include all respective Artifacts in the array of expanded Artifacts. This pattern guarantees

that the Artifacts are expanded even when a page of the list is changed (and therefore new data are loaded).

The pagination of the table is handled by the `Pagination` component.

ParsedArtifactIdentifier component

For the highlighting of the identifier of an Artifact, a `ParsedArtifactIdentifier` component was implemented. The component accepts Artifact as its parameter.

If the Artifact is from a repository type other than NPM or Maven, the component just returns the Artifact identifier with a link to the Artifact detail page. For NPM and Maven Artifact, the identifier is split by a colon character with the JavaScript `split` function. All parts of the identifier, including the colons separating them, are then transformed into PatternFly `Label` components, each having a distinct color. All `Label` components are laid out using PatternFly `Flex` components in one row. Also, a link to the Artifact detail page is included as a link button.

ArtifactsPage component

An `ArtifactsPage` component is the page of Artifacts rendering the `ArtifactsList` component.

7.4.2 Dashboards

For the dashboards, multiple components were created. `ProductMilestoneDetailPage` and `ProductVersionDetailPage` components compose the Product Milestone and Version dashboard pages.

CardFlex component

A `CardFlex` component is a container for data cards. It is implemented as a flex container using the `Flex` PatternFly component positioning data cards in a row. It adds gaps between cards and wraps the row if one row is not enough to hold all the cards. A `CardFlexItem` is a flex container item representing one data card. The component sets the minimum width and height of the card. All cards in one row have the same width, while all cards together span the entire available width of one row by setting the `flex` property of the flex item to 1. It is implemented using the `FlexItem` PatternFly component and also creates a card background with the help of the `ContentBox` component.

The `CardFlexItem` component itself is just a card box. For card value and title, two additional components are meant to be used as children inside it: `CardValue` and `CardTitle`. `CardTitle` holds the title of the card value. `CardValue` displays the actual value corresponding to the title. The titles of cards in one row are aligned vertically using fixed padding and height.

Listing 7.2 portrays the usage of the `CardFlexItem` component in the code.

```
<CardFlexItem>
  <CardValue>{deliveredArtifactsCount}</CardValue>
  <CardTitle>Delivered Artifacts in this Milestone</CardTitle>
</CardFlexItem>
```

Listing 7.2: CardFlexItem component usage

`CardFlexItem` also accepts an optional description as its parameter, which is then displayed using a `BoxDescription` component as a tooltip icon in the upper right corner of the card.

ChartBox component

`ChartBox` is a box that was implemented to be used in specific chart implementations. It accepts a child component as its parameter, which should be a canvas element in which the chart is rendered. The component itself is a div element inheriting width and height from the parent element. It adds some spacing around its child component and centers it via the usage of flex CSS properties. It also accepts the chart description as its parameter and displays it in the form of a tooltip icon in the upper right corner of the box. For description, a `BoxDescription` component is used.

DoughnutChart component

A `DoughnutChart` component uses Chart.js to implement the doughnut chart. The component accepts a list of numbers (doughnut data) and a list of strings (titles corresponding to the data). This is the format of data and labels used by Chart.js. The component returns a canvas element inside the `ChartBox` component. Chart.js uses the canvas element to draw the chart into it.

The component's rendering logic is as follows. On the first render, the chart is created and rendered in the canvas. When the input data are changed, the configuration of the doughnut chart is updated to reflect the new data and the canvas is re-rendered using the Chart.js `update` method. That means that the component is not static and accepts data changes. It also means that data need to be saved inside the state or memoized, lest the chart not be unnecessarily refreshed because the updating logic is inside an `useEffect` hook executed when the chart data changes.

Transformation of Product Milestone statistics

The `DoughnutChart` component accepts the data and labels in a different format than was designed to be returned by the Product Milestone statistics endpoint as portrayed in Section 6.2. To transform the backend data into chart data, a `doughnutChartDataTransform` function was implemented. The implementation of the function is simple and utilizes the JavaScript `Object.values` function to transform the backend data.

A `doughnutChartLabelTransform` function was implemented to transform backend data into the labels using the JavaScript `Object.keys` function.

StackedBarChart component

A `StackedBarChart` implements the stacked bar chart in the Chart.js library. Stacked bars are set to be columns. The stacked bar chart is rendered into a canvas returned by the component and encapsulated by the `ChartBox` component. Its rendering logic is the same as that of `DoughnutChart`, except that it renders a stacked bar chart. The component accepts a list of labels of individual stacked bar columns. Another parameter is data in the format of a list of objects, each containing a label of a characteristic and a list of values of that characteristic in each of the stacked bar columns. This label and data format satisfies the format used by the Chart.js configuration.

Transformation of Product Version statistics

The `StackedBarChart` component also uses a different format of the data and labels than returned by the backend for Product Version statistics. Section 6.3 provides the design of the statistics endpoint. However, the endpoints were designed to be more generic – different chart implementations might use different data formats, and the endpoints were not designed for the specific chart implementation.

A `stackedBarChartDataTransform` function was created to transform backend data into chart data and a `stackedBarChartLabelTransform` function to transform backend data into chart labels. Their implementation is slightly more complex than their doughnut chart variant. The code of the data transformation function is included in Listing 7.3.

```
// statisticsGroup - for example, "artifactQuality"
export const stackedBarChartDataTransform = (data, statisticsGroup) =>
  data &&
  data.length &&
  Object.keys(data[0][statisticsGroup]).map((statisticsName) => ({
    label: statisticsName,
    data: data.map((productMilestoneData) =>
          productMilestoneData[statisticsGroup][statisticsName]),
  }));
```

Listing 7.3: Function transforming Product Version statistics data into a form accepted by the Chart.js stacked bar chart

The function iterates over all characteristics, for example, over all possible Artifact qualities, with the help of the `map` JavaScript method to create an array of objects. For each of these characteristics (`statisticsName`), the object is created. Another `map` method call is used to extract all values of this characteristic in all Product Milestones.

BoxDescription component

A `BoxDescription` component creates an icon that displays the description on the hover event. The icon is implemented as a div element with text aligned to the right. The component supports two variants of the component displayed when hovering over the icon. The first is a tooltip, and the second is a popover.

The component accepts description as its property. If the description is a string, the tooltip implemented using the PatternFly `Tooltip` component is used. In the case where the description is a React component instance, the popover is used that is implemented using the PatternFly `Popover` component. The tooltip displays just plain strings, whereas the popover allows the description to be styled.

ProductMilestoneDetailPage component

A `ProductMilestoneDetailPage` component composes the detail page of a Product Milestone. The PatternFly `Grid` component is used to create the layout of the page. `Grid` component is a grid container that places the layout in the 2D grid and is divided into twelve columns. For a container item, the `GridItem` component is used. The width of the `GridItem` component is set as an integer number of columns it spans. The first grid

container item spanning twelve columns on the page contains the key-value list of Product Milestone details.

The second grid container item holds data cards implemented using the `CardFlex` component. If there were an odd number of data cards, the `GridItem` components themselves would not suffice to make cards of equal width, since twelve is not divisible by an odd number. This is the reason why the `CardFlex` is used within one grid container item that spans the entire twelve columns.

The third and fourth grid container items contain the charts (the `DoughnutChart` component). These grid container items span a responsive number of columns. On small browser windows, each spans twelve columns, and the charts are on top of each other. On large browser windows, each spans six columns, so both charts are displayed side by side.

ProductVersionDetailPage component

A `ProductVersionDetailPage` component represents the Product Version detail page. It uses the same layout structure as the `ProductMilestoneDetailPage` component. It only displays different data and stacked bar charts (the `StackedBarChart` component) instead of doughnut charts. The endpoints for the Product Version charts are paginated; therefore, the `Pagination` component allows users to navigate the pages.

7.4.3 Network graphs

For the network graphs, one hook was created. The Product Milestone interconnection graph page is contained in a `ProductMilestoneInterconnectionGraphPage` component and the Build Artifact dependency graph in a `BuildArtifactDependencyGraphPage` component.

useNetworkGraph custom hook

A `useNetworkGraph` is a custom hook for the creation and management of the state of the network graph and its rendering. The ID of the div container into which the graph will be rendered is passed to the hook. The hook returns a `createNetworkGraph` function which creates the graph and renders it inside the div container mentioned above. The graph data structure is created and managed by the Graphology library. The rendering part is handled by the Sigma.js library. The creation function accepts a callback as its parameter. This callback is called inside the creation function once the graph is created. The purpose of the callback is to add nodes and edges to the graph. Therefore, the set of nodes and edges is not directly passed to the hook. The callback also sets the color and labels of the nodes and edges. After this callback is run, the graph is laid out using the ForceAtlas2 algorithm, which is part of the Graphology library.

In addition, the name of the main node is passed to the component. Relative to this main node, nesting level functionality is applied. To apply the nesting level, the breadth-first graph search algorithm[1] is used. It traverses the whole graph of nodes, and nodes, whose distance is larger than the set nesting level, are hidden. The algorithm is run inside the `useEffect` hook each time the nesting level is changed.

The `useNetworkGraph` hook states include hovered node, dragged node, selected edge, and a set of selected nodes. These states are managed by the Sigma.js event handlers

[1] https://en.wikipedia.org/wiki/Breadth-first_search

that were created by the `createNetworkGraph` function. Some states of the graph are inputted into the hook from outside since they are managed by outside components. These include primary filter text (managed by the search bar component), secondary filter text (managed by the search bar too), and the nesting level (managed by toggle and number input components). All of these states (internal or external) modify the visualization of nodes and edges. To reflect the current states, the Sigma.js node and edge reducers are executed in the `useEffect` hook. Dependencies of this `useEffect` hook include the states mentioned. The node and edge reducers then iterate over all nodes and edges and apply visual changes appropriate to the states; for example, the selected nodes are highlighted.

The ForceAtlas2 layout algorithm is managed by its Webworker Graphology implementation. The layout is controllable and can be started or stopped. The hook returns the function to start and the function to stop the layout algorithm. This allows the parent component to control the algorithm.

ProductMilestoneInterconnectionGraph component

A `ProductMilestoneInterconnectionGraph` component is responsible for the UI of the graph. The component renders a div container in which the Sigma.js graph is rendered. The management of the graph data structures and rendering is done by the `useNetworkGraph` hook. The creation of a graph is achieved with the help of the `createNetworkGraph` function returned by the mentioned hook. The component accepts graph data as its parameter; once the data are loaded, the `createNetworkGraph` function is called to create the graph. Into the `createNetworkGraph` function, a callback is passed, adding nodes and edges contained in the input data into the graph.

The component also displays the button that controls the graph layout algorithm. The component receives strings of search bars and the nesting level states via parameters which are then passed to the mentioned hook.

ProductMilestoneInterconnectionGraphPage component

A `ProductMilestoneInterconnectionGraphPage` component composes the whole Product Milestone interconnection graph page. The `ProductMilestoneInterconnectionGraph` component renders the graph.

The `ArtifactsList` renders the list of shared Delivered Artifacts between the Product Milestones and the list of displayed columns is configured to include columns that were designed to be here in the corresponding wireframe.

The component manages the state of search bars and nesting level input. The search bars are implemented by the `SearchInput` PatternFly component. The nesting level is toggled by the `Switch` PatternFly component. The `NumberInput` PatternFly component handles the nesting level value.

BuildArtifactDependencyGraph component

A `BuildArtifactDependencyGraph` component creates the UI of the network graph of the Build Artifact dependencies. The structure of the component is the same as that of the `ProductMilestoneInterconnectionGraph` component.

BuildArtifactDependencyGraphPage component

A `BuildArtifactDependencyGraphPage` component composes the Build Artifact dependency graph page. It follows the same structure as its Product Milestone interconnection graph counterpart. The `ArtifactsList` component renders the list of Artifact dependencies.

7.4.4 Product Milestone comparison

The Product Milestone comparison is implemented in one table component. Page component containing the table is `ProductMilestoneComparisonPage`.

ProductMilestoneComparisonTable component

A `ProductMilestoneComparisonTable` is a table of Product Milestone Comparison. The table is implemented by the `TableComposable` PatternFly component. The `SearchSelect` components provide the tree select elements which are used to find the Product, Product Version, and Product Milestone, respectively.

A service container that fetches the data of the table is passed to the component. At first, only three mentioned select elements, and the "Add Column" and "Fetch" buttons are displayed. Once a Product Milestone is selected in the appropriate select element, the "Add Column" button can be used to add a column of a Product Milestone to the table. When the first Product Milestone is added, a table header is displayed. The "Fetch" button fetches the data of the table with the passed service container. The list of IDs of selected Product Milestones is passed to the `run` function of the service container. Once the data are loaded, the component iterates over the fetched data with the `map` JavaScript method and displays the individual rows, each containing versions of a Delivered Artifact in selected Product Milestones.

The `InnerScrollContainer` PatternFly component encapsulates the table. It creates a horizontal scrollbar if all columns do not fit into the table.

ProductMilestoneComparisonPage component

A `ProductMilestoneComparisonPage` component creates the page of the Product Milestone comparison with the help of the `ProductMilestoneComparisonTable` component.

7.5 Testing and feedback

The new UI project is currently in development and it is not yet in production. It only has a development environment (the master environment). Also, the backend for the designed REST API was not yet implemented.

Because of these reasons, the work of this Bachelor's thesis was not yet tested by the users of the PNC system. Instead, feedback was provided by the PNC team. The code was developed in the GitHub repository using the GitHub flow[2]. Therefore, the code was added to the repository through Pull Requests. Each Pull Request of this thesis was reviewed by other developers. This includes the code review.

Throughout the implementation process, the work of the thesis was consulted with members of the PNC team. After the code was merged into the repository, the code was tested manually in the master environment. If any bugs were found during the testing, or if any UI features were visibly missing, new Pull Requests were created to address these issues. Some of the developers reported some small problems, such as the format of tooltip descriptions. Also, the author of this very thesis presented a demonstration of the implemented UI components to the members of the team.

Overall, six members of the PNC team shared their opinions and provided feedback during or after the development of the thesis. Specifically, feedback was provided by the Product Owner[3] of the PNC project, who oversaw the development of this Bachelor's thesis and oftentimes presented user requirements.

> *"PNC build system contains lots of various information about products and dependencies between those products. The system has the information, but it is not easily readable in the current form. Patrik's work helps with the visualization of the data, highlighting various statistics and discovering dependencies between products, which might be sometimes hidden even to its developers. The work meets my expectations and it delivers new features, which were requested by PNC users."*
> — Ing. Jakub Barteček, Manager, Software Engineering, PNC Product Owner

PNC UI lead, who helped with the UI aspect of the thesis, also offered his views.

> *"The current version of PNC provides no way to visualize Product-related data, forcing users to manually explore the available data to get a better overview and find important relationships between individual entities. Patrik's work successfully and significantly improved the current state. For example, I would like to highlight that the dashboards are well structured, the readability of the Artifact list is improved and the network graphs provide a unique and easy-to-navigate way of displaying the required information. Patrik also listened and responded to feedback, resulting in an improved solution."*
> — Mgr. Martin Kelnar, Senior Software Engineer, PNC UI lead

[2] https://docs.github.com/en/get-started/quickstart/github-flow
[3] https://www.scrum.org/resources/what-is-a-product-owner

Chapter 8

Conclusion

The goal of this Bachelor's thesis was to enhance the visualization of Product-related pages on the new UI of the PNC build system, namely the visualization of relationships between PNC entities, including Products, Product Versions, Product Milestones, Builds, and Artifacts. This goal was achieved successfully.

Firstly, the TypeScript language and multiple libraries were studied. The libraries including React, Chart.js, Sigma.js, and Graphology, were explored, mainly from their documentation. PNC build system was analyzed including its entities and their relationships. Internal PNC documentation and discussion with developers of the system, but also experience with the usage of the system, helped with learning about the PNC system. Also, Product-related pages on the original PNC UI were examined.

Then, user requirements were acquired regarding the enhancements of the related UI, but developers also provided feedback. With all of this information, wireframes were created to illustrate new designs. The new visualization was designed in the form of tables, charts, network graphs, and dashboards. For the designed pages, REST API was designed, but since the implementation of endpoints was not part of this Bachelor's thesis, designed endpoints were not implemented. The implementation of the pages was fulfilled using the language and libraries mentioned. Lastly, a video was recorded that demonstrates the newly implemented UI components.

Overall, one page was redesigned, two pages were extended, and three new pages were created. The Artifacts list is now more readable, mainly because of the Artifact identifier parsing option. The Product Milestone and Product Version detail pages were extended into dashboards of statistics and charts. Two network graphs were created. The first is the Product Milestone interconnection graph visualizing sharing of Delivered Artifacts between Product Milestones. The second is the Build Artifact dependency graph displaying build-time dependencies between Builds. The last implemented component is the Product Milestone comparison which allows users to compare Delivered Artifacts of selected Product Milestones. All code was contributed to an open-source repository on GitHub. The whole work of this Bachelor's thesis is composed of 20 GitHub Pull Requests.

The newly created code provides a way to implement other visualizations of this kind since the code is easily expandable and editable. The UI components form a basis for future needs. For example, a new network graph to display build-time dependencies between Product Milestones could be added, or dashboards could be created for other PNC entities, or existing ones could be extended with new charts.

Bibliography

[1] Downie, N. *Chart.js documentation* [online]. April 2023 [cit. 2023-05-02]. Available at: https://www.chartjs.org/docs/latest/.

[2] Jacomy, M., Venturini, T., Heymann, S. et al. ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software. *PLOS ONE* [online]. 2014, vol. 9, no. 6, [cit. 2023-05-03]. Available at: https://doi.org/10.1371/journal.pone.0098679.

[3] MDN. *Using Web Workers* [online]. [cit. 2023-05-03]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers.

[4] Meta Platforms, Inc. *React documentation* [online]. 2023 [cit. 2023-04-22]. Available at: https://react.dev/.

[5] Microsoft. *TypeScript documentation* [online]. 2023 [cit. 2023-05-01]. Available at: https://www.typescriptlang.org/docs/.

[6] npm. *npm Docs: package.json* [online]. [cit. 2023-04-29]. Available at: https://docs.npmjs.com/cli/v9/configuring-npm/package-json.

[7] npm. *npm Docs: scope* [online]. [cit. 2023-04-29]. Available at: https://docs.npmjs.com/cli/v9/using-npm/scope.

[8] Plique, G. *Graphology, a robust and multipurpose Graph object for JavaScript* [online]. Zenodo, 2023 [cit. 2023-05-01]. Available at: https://doi.org/10.5281/zenodo.5681257.

[9] PNC development team. *Project Newcastle Orchestrator GitHub repository* [online]. [cit. 2023-04-29]. Available at: https://github.com/project-ncl/pnc.

[10] PNC development team. *Project Newcastle React UI Github repository* [online]. [cit. 2023-05-02]. Available at: https://github.com/project-ncl/pnc-web-ui-react.

[11] Red Hat, Inc. *Patternfly* [online]. 2022 [cit. 2023-05-02]. Available at: https://www.patternfly.org/v4/.

[12] Schoger, S. and Wathan, A. *Refactoring UI.* Self-published, 2018.

[13] Sigma.js development team. *Sigma.js GitHub repository* [online]. [cit. 2023-05-01]. Available at: https://github.com/jacomyal/sigma.js.

[14] The Apache Software Foundation. *Guide to naming conventions on groupId, artifactId, and version* [online]. 27. april 2023 [cit. 2023-04-29]. Available at: https://maven.apache.org/guides/mini/guide-naming-conventions.html.

[15] The Apache Software Foundation. *POM Reference* [online]. 27. april 2023 [cit. 2023-04-29]. Available at: https://maven.apache.org/pom.html.

Appendix A

Contents of the included storage media

The storage media has the following structure.

- `git-patch-files/` – git patch files of the source code implemented in this Bachelor's thesis
- `technical-report-source/` – source code of the technical report of this Bachelor's thesis
- `xkoryt04-technical-report.pdf` – normal version of the technical report of this Bachelor's thesis
- `xkoryt04-technical-report-print.pdf` – print version of the technical report of this Bachelor's thesis
- `demo-video.mkv` – demonstration video of the implementation part of this Bachelor's thesis
- `README` – file describing the storage media

Appendix B

GitHub Pull Requests

The following GitHub Pull Requests implement features of this Bachelor's thesis.

- https://github.com/project-ncl/pnc-web-ui-react/pull/166
- https://github.com/project-ncl/pnc-web-ui-react/pull/170
- https://github.com/project-ncl/pnc-web-ui-react/pull/173
- https://github.com/project-ncl/pnc-web-ui-react/pull/176
- https://github.com/project-ncl/pnc-web-ui-react/pull/183
- https://github.com/project-ncl/pnc-web-ui-react/pull/184
- https://github.com/project-ncl/pnc-web-ui-react/pull/186
- https://github.com/project-ncl/pnc-web-ui-react/pull/187
- https://github.com/project-ncl/pnc-web-ui-react/pull/189
- https://github.com/project-ncl/pnc-web-ui-react/pull/190
- https://github.com/project-ncl/pnc-web-ui-react/pull/191
- https://github.com/project-ncl/pnc-web-ui-react/pull/195
- https://github.com/project-ncl/pnc-web-ui-react/pull/198
- https://github.com/project-ncl/pnc-web-ui-react/pull/199
- https://github.com/project-ncl/pnc-web-ui-react/pull/200
- https://github.com/project-ncl/pnc-web-ui-react/pull/201
- https://github.com/project-ncl/pnc-web-ui-react/pull/206
- https://github.com/project-ncl/pnc-web-ui-react/pull/210
- https://github.com/project-ncl/pnc-web-ui-react/pull/211
- https://github.com/project-ncl/pnc-web-ui-react/pull/220

Appendix C

Final Results

In this appendix, the final results of the implemented Product-related pages on the new UI of the PNC build system are shown. Data on them are not real, they are mocked.

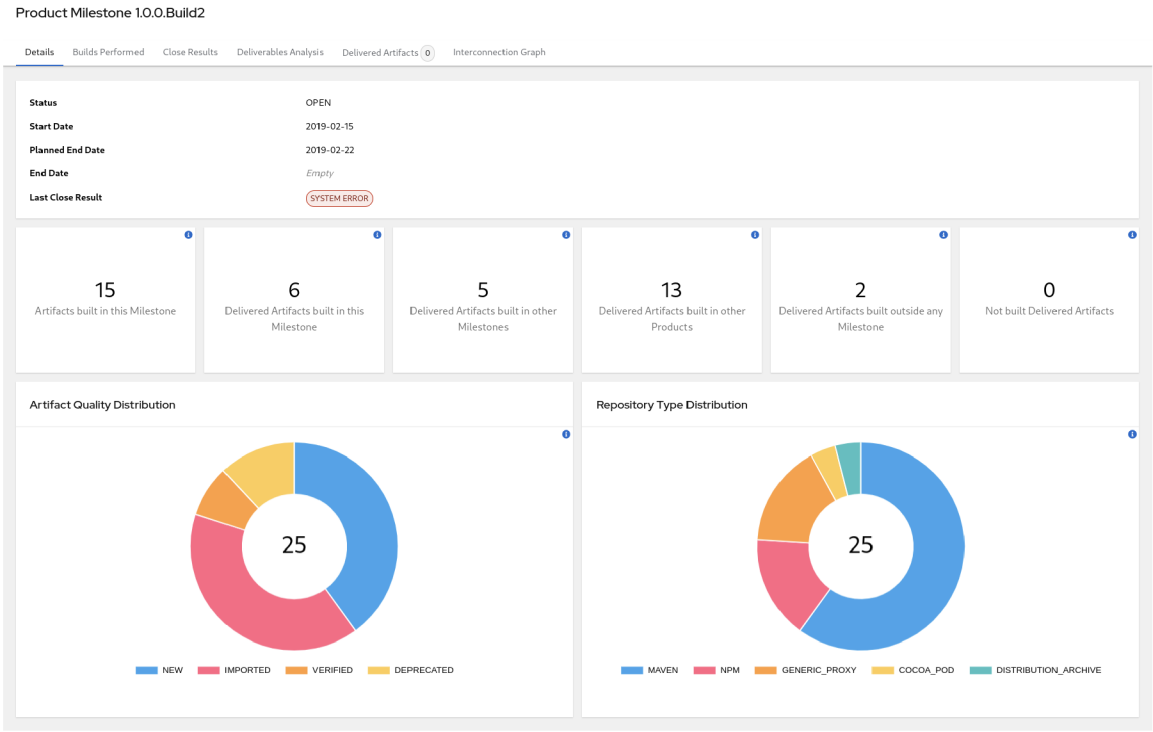

Figure C.1: Final result of the Product Milestone dashboard

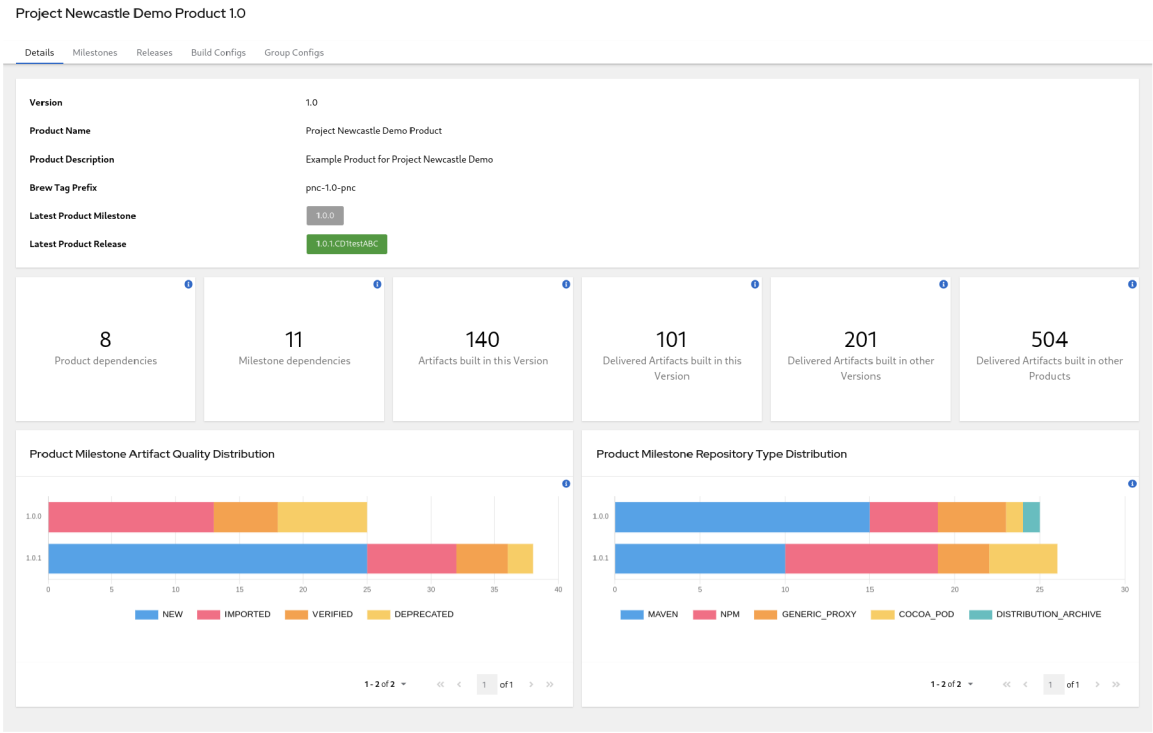

Figure C.2: Final result of the Product Version dashboard

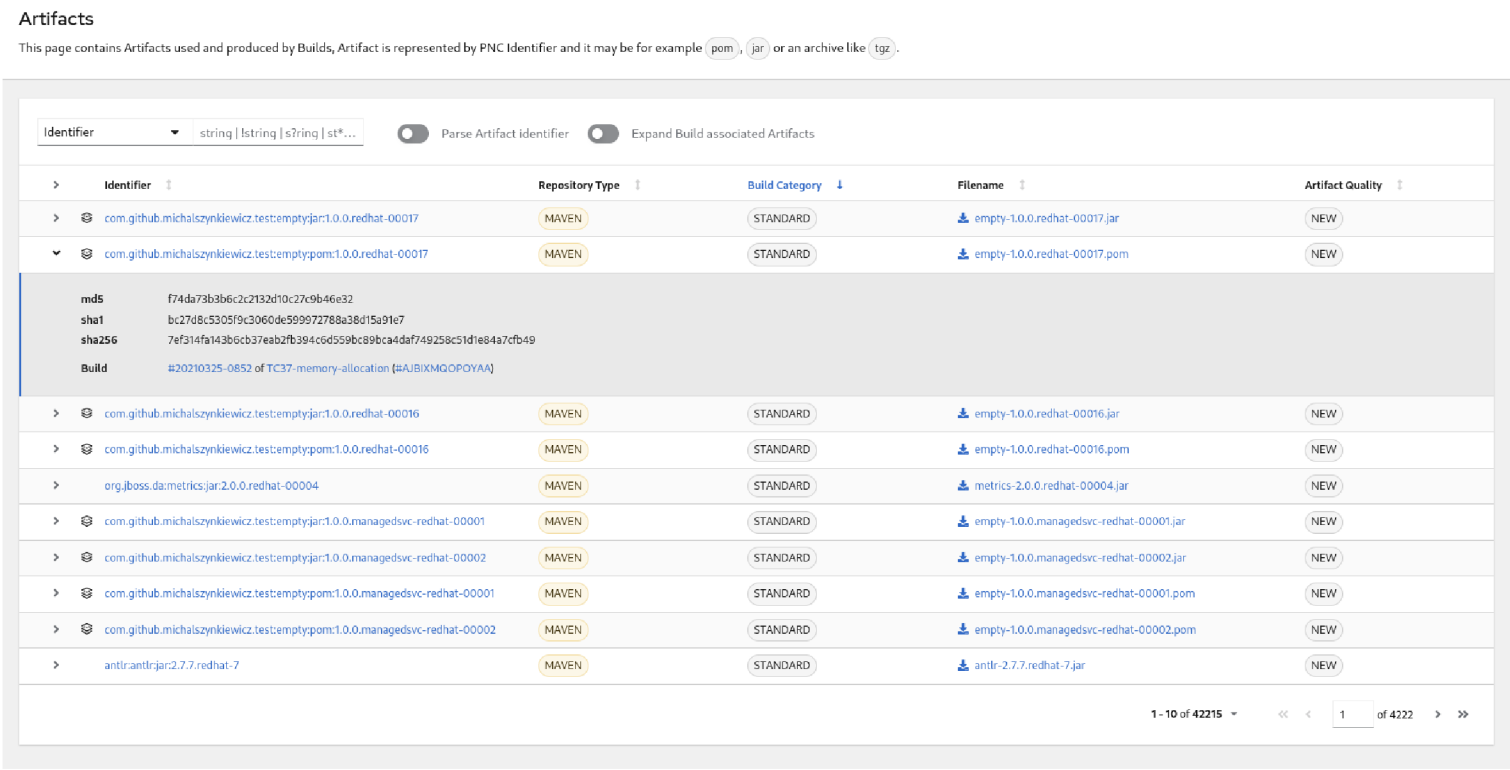

Figure C.3: Final result of the Artifacts list

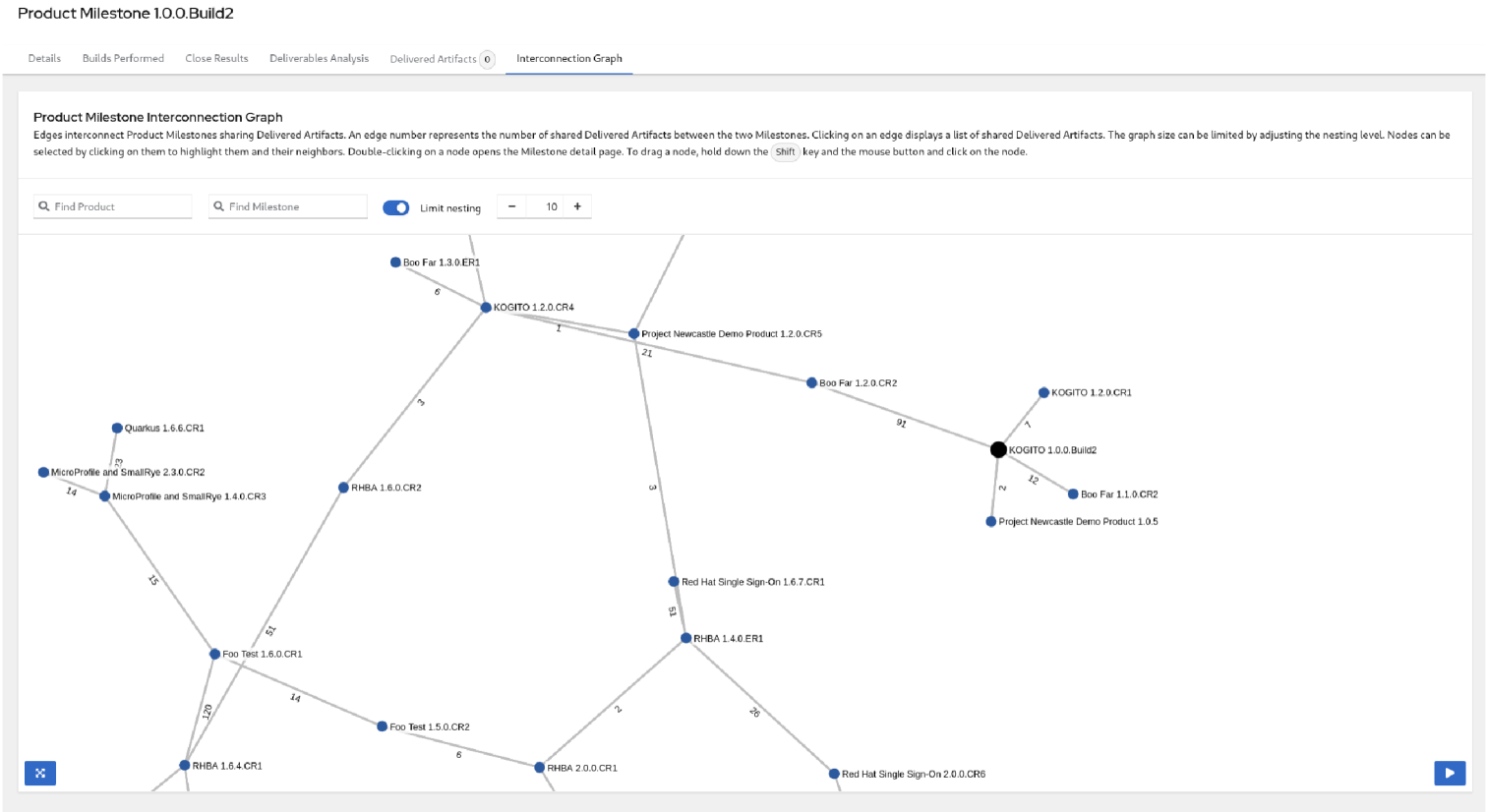

Figure C.4: Final result of the Product Milestone interconnection graph

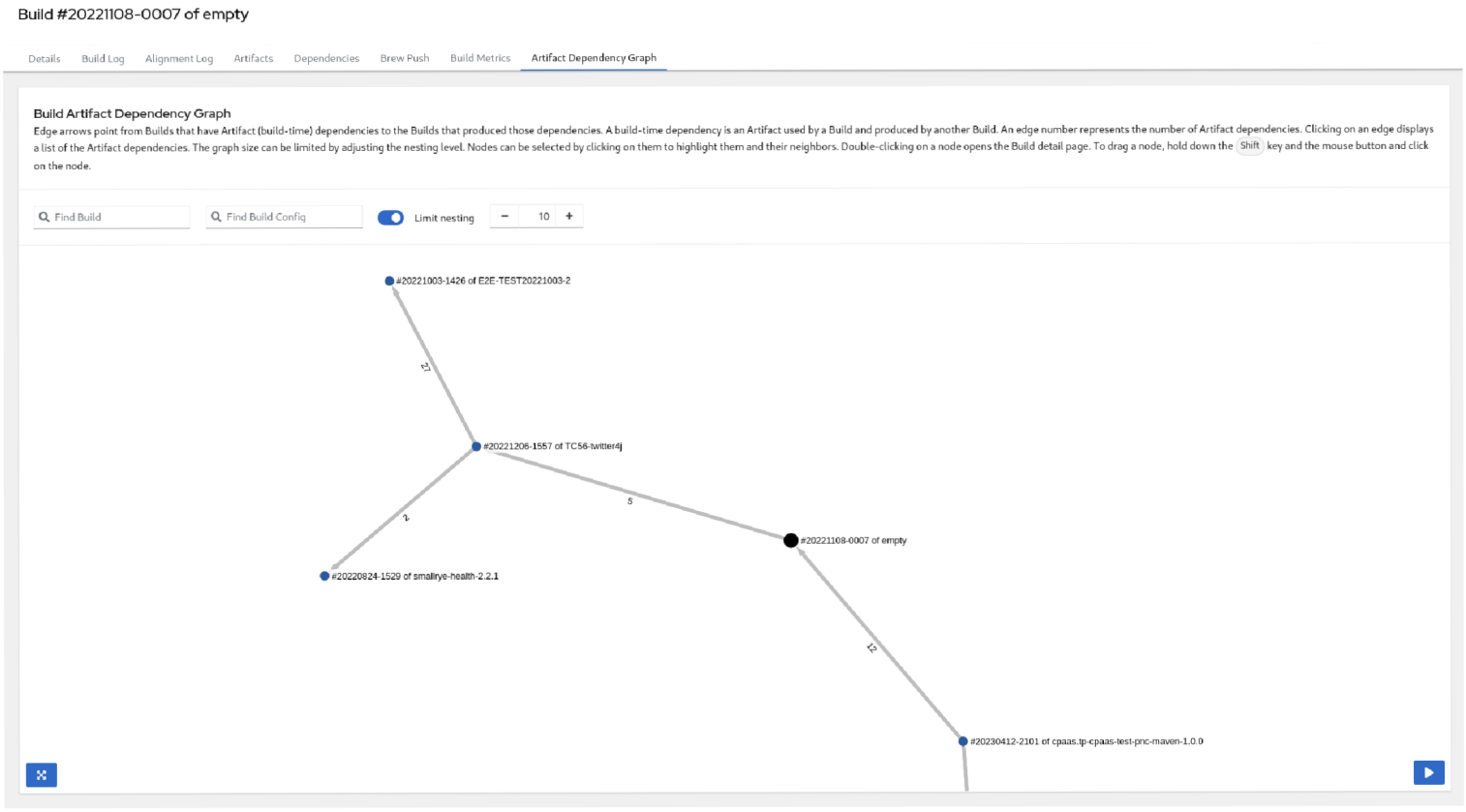

Figure C.5: Final result of the Build Artifact dependency graph

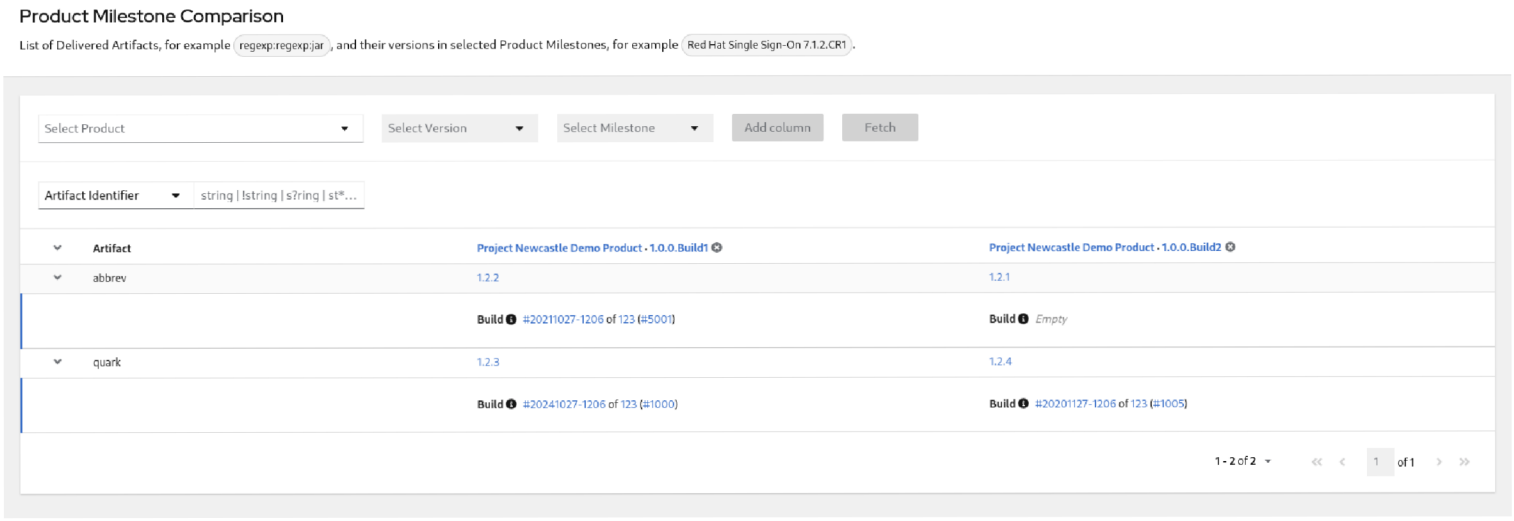

Figure C.6: Final result of the Product Comparison table