“Building the Next Generation Personal Data Platforms”

G.A. n. 871370

Deliverable 2.3

Release of tools to improve user’s privacy

H2020-EU-2.1.1: **PIMCity**

Project No. 871370

Start date of project: 01/12/2019

Duration: 33 months

Revision: 01

Deliverable delivery: 28/02/2022

Deliverable due date: 28/02/2022

Document Information

Document Name: Release of tools to improve user's privacy

WP2 Title: Tools to improve data subjects' privacy

Task 2.1, 2.2, 2.3, 2.4

Revision: 01

Revision Date: 15/02/2022

Authors: POLITO and all WP2 partners

Dissemination Level

Project co-funded by the EC within the H2020 Programme		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Approvals

	Name	Entity	Date
WP Leader	Martino Trevisan	POLITO	15/2/2022
Author	Luca Vassio	POLITO	15/2/2022
Author	Nikhil Jha	POLITO	15/2/2022
Author	Stefano Traverso	ERMES	15/2/2022
Author	Davide Pozza	ERMES	15/2/2022
Author	Rodrigo Irarrazaval	WIBSON	15/2/2022
Author	Daniel Fernandez	WIBSON	15/2/2022
Author	Javier Calvo	WIBSON	15/2/2022
Author	Roberto Gonzalez	NEC	15/2/2022
Author	Daniel Oñoro	NEC	15/2/2022

Author	**Bhushan Kotnis**	**NEC**	**15/2/2022**
Reviewer	**Roberto Gonzalez**	**NEC**	**15/2/2022**
Coordinator	**Marco Mellia**	**POLITO**	**15/2/2022**

Document history

Revision	Date	Modification
Version 1	15/02/2022	V1

List of abbreviations and acronyms

Abbreviation	Meaning
PIMS	Personal Information Management System
PDK	PIMS Development Kit
TT	Transparency Tags
P-DS	Personal-Data Safe
P-PM	Personal-Privacy Metrics
P-CM	Personal-Consent Manager
P-PPA	Personal-Privacy Preserving Analytics

Executive Summary

This deliverable includes the User guides of the components devoted to improving the user's privacy in the PIMCity PDK. The final version of the tools is delivered along with this deliverable. The source code of the PDK modules is available on GitLab, under the PIMCity/WP2 project[1], and we next provide the URLs for each repository.

This document covers of the following PDK modules:

1. Personal Consent Manager (P-CM)
2. Personal Privacy Metrics (P-PM):
3. Personal Privacy Preserving Analytics (P-PPA):
4. Personal Data Safe (P-DS):

An overview on the PIMCity PDK can be found in deliverable D1.1. The design of these modules is described in detail in Deliverable 2.2, along with the motivation behind our design choices and their overall architecture. More details on the design of other modules are available in deliverables D3.3 and 4.2. Together with this Deliverable, we publish the user guides of the other PDK modules in deliverables D3.3 and D4.2.

[1] https://gitlab.com/pimcity/

Index

1.- Introduction and Deliverable Objectives

Thanks to the introduction of regulatory frameworks focused on user's privacy such as EU's General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA), we are testifying the diffusion of new systems whose purpose is to help users in storing, understanding, and, possibly, monetizing their personal data in a transparent and easy way.

In this context, WP2 aims at building a set of software modules with the goal of enhancing the users' privacy. To this end, we design components to allow users store their data in a secure way. Second, users must be provided with tools to let them control and manage consent in a transparent way, i.e., decide who, how and when can access data. Third, we aim at creating tools for allowing queries on data that respect the users' consent choices and allow privacy-preserving data analyses. Finally, it is fundamental to present detailed information about the services willing to access data (i.e., data buyers) and the purposes of their business.

In the Work Package 2 of the PIMCity project, we target the goals described above and aim at designing and implementing modules that accomplish them. To this end, we developed various module, as part of the **PIMCity PIMS Development Kit (PDK),** basic and generic components that offer fundamental functionalities for Personal Information Management System (PIMS). These modules aim at empowering the users to control how their data is stored, processed, shared, and for which purposes. The modules described in this deliverable are:

1. **Personal Consent Manager (P-CM):** The consent manager is the means to define all the user's privacy preferences. It defines which data a service is allowed to collect, process, or which can be shared with third parties by managing explicit consent. Users' settings are imposed on all participating systems. The P-CM is described in Section 2 and is available online as an open-source project at: https://gitlab.com/pimcity/wp2/personal-consent-manager
2. **Personal Privacy Metrics (P-PM):** They have the goal of increasing the user's awareness. They collect, compute and share easy to understand novel privacy metrics, indicating e.g., which information the system is collecting, how it stores and manages the data, if it shares it with third parties. This module is described in Section 3 and is available online as an open-source project at: https://gitlab.com/pimcity/wp2/privacy-metrics
3. **Personal Privacy Preserving Analytics (P-PPA):** This module has the goal of allowing data analysts and stakeholders to retrieve useful information from the data, while preserving the privacy of the users whose data are in the studied datasets. It leverages concepts like Differential Privacy and K-Anonymity so that data can be exchanged among different systems while preserving the actual information as private. It is described in Section 4 and is available online as an open-source project at: https://gitlab.com/pimcity/wp2/personal-privacy-preserving-analytics
4. **Personal Data Safe (P-DS):** It is the means to store personal data in a controlled form. It implements a secure repository for the user's personal information like navigation history, contacts, preferences, personal information, etc. It is described in Section 5 and is available online as an open-source project at: https://gitlab.com/pimcity/wp2/personal-data-safe

In this deliverable, we include the User's guide of the PDK these. For each module we report detailed instructions for installing and using it. We precisely list all modules' requirements and outline their operation. The full architectural design of the modules, however, is described in detail in Deliverable 2.2. Along with this deliverable, we release the final implementations of the Work Package 2 PDK modules. We defer the reader to Deliverable 1.1 for the full requirements for all modules. This deliverable *does not* cover module integration and the overall system design, that is covered in WP1 and WP5 deliverables.

The objectives of this deliverable partially address the following objectives of WP2 described in the Grant Agreement:

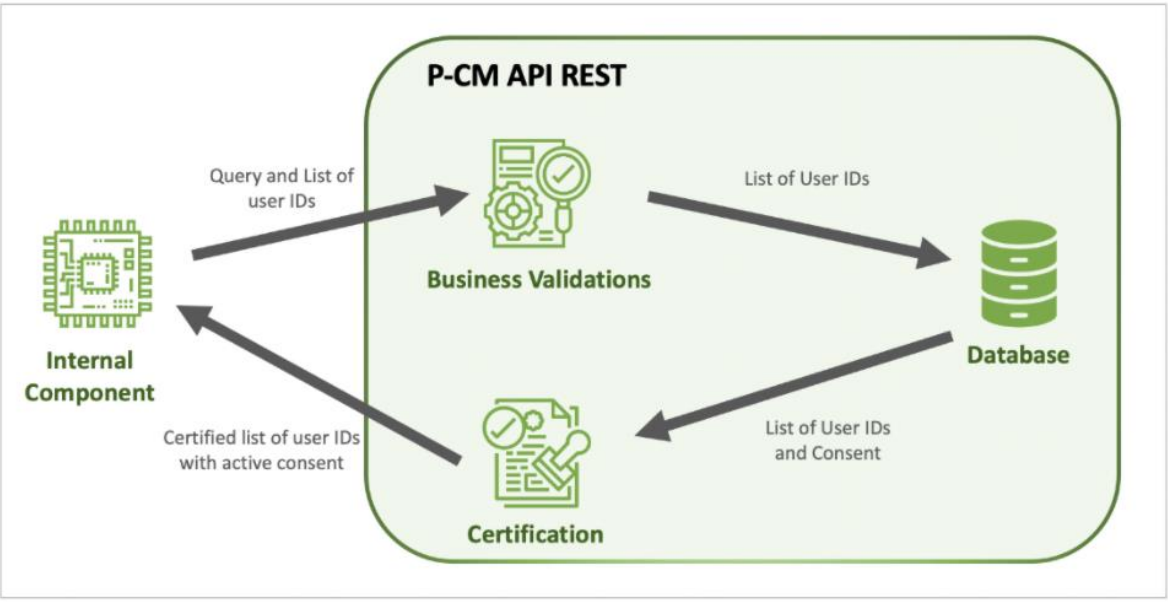
- *Design and develop a system able to empower the users to control their consent settings in multiple account and services. It should have an easy-to-use interface and provide auto configuration options to make it easier for the users to configure complex scenarios by using aggregated/crowdsourced data of the different users to build a set of common profiles.*
- *Design the Privacy Metrics to i) unveil and communicate end users the data collected by online services, ii) automatically identify and pinpoint possible privacy violations in data collection, iii) communicate these findings to the end users in an easy and intuitive user interface.*
- *Develop a set of general-purpose building blocks to analyze users' data without affecting their privacy. It will offer some algorithms and methodologies able to provide a certain level of anonymity using concepts as zero-knowledge proof or k-anonymity.*

Finally, we remark that further adjustment to the current implementation of modules presented in this document might occur during the remaining execution of the project. In particular, during the PIMCity deployment initiatives, we will potentially modify the modules to include features needed for demonstration purpose and/or collect bugs. However, we will avoid revolutionizing the solutions presented in this document, and possible changes will be described in future deliverables, or directly in components' code repositories.

2.- Personal Consent Manager (P-CM)

Overview

The primary objective of the Personal Consent Manager (P-CM) is to give the users the transparency and control over their data in a GDPR compliant way. That is, give them the possibility to decide which data can be uploaded and stored in the platform, as well as how (raw, extracted or aggregated) data can be shared with Data Buyers in exchange for value when the opportunity arises. The P-CM is presented as a web application and a REST API, not only providing users the possibility to use the component in a user-friendly way, but also enabling developers to integrate PIMCity Consent Management capabilities in their products. The architecture of the PDK is depicted in the figure below.



Installation

The documentation and the following instructions refer to a Linux environment, running with **Docker Engine v20.10.x** and **Docker Compose: v1.27.x**. The P-CM project has been cloned from the GitLab repository at https://gitlab.com/pimcity/wp2/personal-consent-manager.

Follow accurately the next steps to quickly set-up the P-CM backbone on your server. All relevant steps are designed for a Linux machine, perform the equivalent procedure with other environments that support Docker.

Prepare the environment:

```
> sudo apt-get update
> sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
> curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --
dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
> echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-
keyring.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
> sudo apt-get update
> sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Import the project from the GIT repository:

```
> git clone https://gitlab.com/pimcity/wp2/personal-consent-manager.git
```

Usage

Execution:

To run the P-CM service, the following command should be executed at the root of the repository:

```
> docker-compose up
```

Configuration:

The P-CM can be configured in three similar ways:

1. Using environment variables.
2. Defining those environment variables in a file called ".env" at the root directory of the folder with the PDK deployed, declaring them in the same way a variable is defined in the UNIX shell. You can check the example in `backend/.env.example`.
3. Modifying docker-compose.yml file in this repository.

Usage Example:

You can check the Swagger OpenAPI definition to see how the API is defined and used at: https://easypims.pimcity-h2020.eu/pcm-api/api-docs/. Examples are provided as well.

Authentication:

KeyCloak service is used to carry out the authentication process. The user will not have to log in directly to the P-CM, but provide a JWT obtained from the authentication service or an EasyPIMS component, such as the PDA.

Data Structure:

For each data category and data sharing purpose, the user can enable or disable a specific consent. Therefore, the data structure used by the P-CM is essentially the tuple (`USER_ID, DATA_CATEGORY, DATA_SHARING_PURPOSE, IS_ACTIVE`). More information is added to the consent, such as timestamps and so on, to provide an accurate service.

Changes

- 29 December 2021: added Oauth integration
- 29 December 2021: added OpenAPI UI
- 21 January 2022: improved default consent setting
- 23 January 2022: improved DynamoStore ORM component

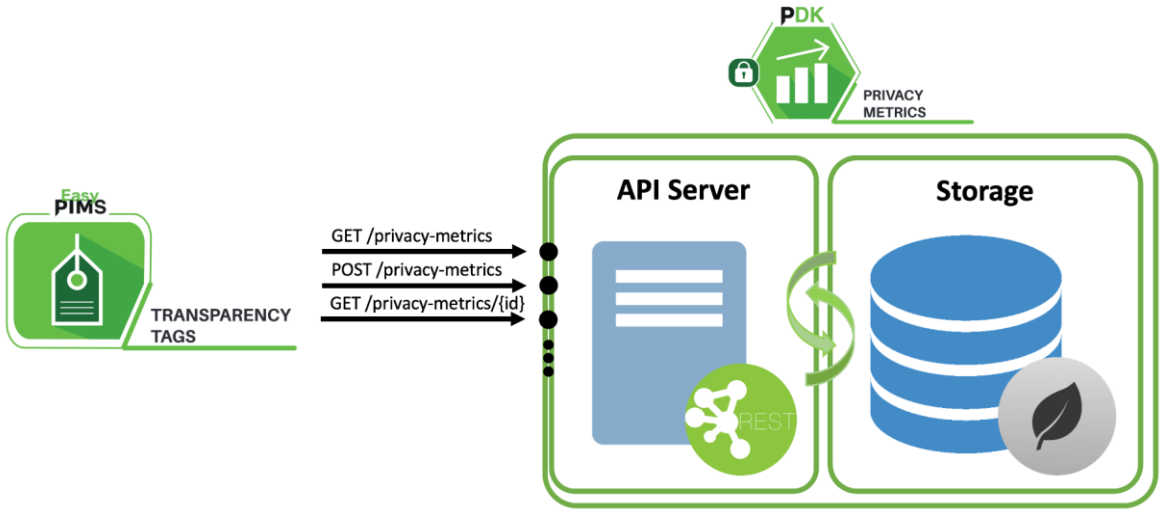
3.- Personal Privacy Metrics (P-PM)

Overview

Privacy Metrics represent the means to increase the user's awareness. This component collects, computes and shares easy-to-understand data to allow users know how a service (e.g., a data buyer) stores and manages the data, if it shares it with third parties, how secure and transparent it looks, etc. These are all fundamental pieces of information for a user to know to take informed decisions. The PM computes this information via a standard REST interface, offering an open knowledge information system which can be queried using an open and standard platform. PMs combine information from supervised machine learning analytics, services themselves and domain experts, volunteers, and contributors. The Open API implementation of Privacy Metrics component is available at official PIMCity's Gitlab code repository, at the address: https://gitlab.com/pimcity/wp5/open-api/-/blob/master/WP2/privacy-metrics.yml. For a complete description of the data contained in Privacy Metrics, refer to Sec. 4.3.1 of PIMCity's Deliverable D2.2.

Privacy Metrics implements authorization based on OAuth2.0 model using Client Credentials flow.

In this repo we provide the implementation of the backend offering authenticated access to PMs. It builds on MongoDB (for the database), Python/Flask and Swagger (for the server). The repo builds on Poetry for the management of Python packaging and dependencies.



Installation

Native Deployment

Pyenv, Python and Poetry

We recommend to install pyenv to install and manage different Python versions: Install pyenv using the guide at: https://github.com/pyenv/pyenv

Then, install Python3.9. In the following, we are using the latest version available (3.9.5) at the moment of this writing.

```
pyenv install 3.9.5
```

Set the installed version as global:

```
pyenv global 3.9.5
```

Now, install Poetry using this guide.

Configure Poetry to use local virtual environments:

```
poetry config virtualenvs.in-project true
```

Then, install the project dependencies with poetry:

```
# Install dependencies
poetry install
```

Note: You can switch back to previous Python version with:

```
# Switch back to system Python
pyenv global system
```

MongoDB

On macOS, we recommend to install MongoDB with Brew (https://brew.sh/):

```
brew install mongodb-community
```

On Ubuntu, you can install MongoDB with apt:

```
sudo apt install mongodb
```

Alternatively, you can install MongoDB using the guide at: https://docs.mongodb.com/manual/installation/.

Configuration and Execution

First, start MongoDB and import data. In a terminal, run the commands listed below.

macOS:

```
# Start MongoDB
brew services start mongodb/brew/mongodb-community
```

Ubuntu:

```
sudo service mongodb start
```

Once launched, populate the DB with Privacy Metrics data:

```
# Import data
mongoimport --db testdb --collection privacy_metrics --drop --file
seed/privacy_metrics.json –jsonArray
```

Specify the IP address and port on which MongoDB is listening

```
export MONGO_HOST=localhost
export MONGO_PORT=27017
```

Set environment and debug mode on if needed (default is off):

```
export FLASK_ENV=development
export SERVER_DEBUG=True
```

Then, run the API server

```
# Run the API server
poetry run python3 -m privacy_metrics.main
```

Deployment with Docker

This repo comes with the gears necessary to deploy the Privacy Metrics module in a dockered environment.

Pyenv, Python and Poetry

We recommend to install pyenv to install and manage different Python versions:

Install pyenv using this guide: https://github.com/pyenv/pyenv

Then, install Python3.9. In the following, we are using the latest version available (3.9.5) at the moment of this writing.

```
pyenv install 3.9.5
```

Set the installed version as global:

```
pyenv global 3.9.5
```

Now, install Poetry using this guide.

Configure Poetry to use local virtual environments:

```
poetry config virtualenvs.in-project true
```

Docker

On macOS, we recommend to install Docker using this guide:
https://docs.docker.com/docker-for-mac/install/

On Ubuntu, you can install Docker with snap

```
sudo snap install docker
```

Configuration and Execution

First, launch Docker daemon.
On macOS the daemon is started in the background together with Docker Desktop app.
On Ubuntu, launch the daemon with the following command:

```
sudo snap start docker
```

Specify the IP address and port on which MongoDB is listening

```
export MONGO_HOST=mongo
export MONGO_PORT=27017
```

Set user and group IDs:
```
export DUID=$UID
export DGID=`id -g`
```

Set environment and debug mode on if needed (default is off):
```
export FLASK_ENV=development
export SERVER_DEBUG=True
```

Then, run the command below.
```
# Build and run dockerized Privacy Metrics
./build.sh
```

To stop the services use "CTRL+C" and run the command below.
```
# Stop and destroy containers
./stop_and_destroy.sh
```

Authorization

This module comes with a setup to integrated with PIMCity's EasyPIMS deployment which uses a on-premise KeyCloak instance as authentication provider. For using Privacy Metrics in your environment with OAuth2.0 authorization enabled still using KeyCloak, you have to modify `tokenUrl` parameter in `swagger.yaml` the KeyCloak client configuration defined in `authorization_controller.py`. Differently, to enable authorization workflow with other types of authentication providers, you must implement your own setup in `authorization_controller.py`.

Usage

Swagger UI

NOTE: Swagger UI is available in debug mode only (`SERVER_DEBUG=True`)

Open your browser on `http://localhost:8080/privacy-metrics/ui`. This page provides the Swagger UI describing Privacy Metrics APIs, together with actions to activate and test APIs.

In order to test the APIs, you have to get authorization:

- on the UI page click on `Authorize`. In the modal `Available Authorizations` scroll down to `OAuth2 (OAuth2, clientCredentials)`, and fill the form with the client ID and secret that have been provided by the authorization provider administrator.
- select the scopes for which the client ID has been enabled.
- click on `Authorize` button.

Once obtained authorization to interact with APIs, Swagger UI will enable you to test all APIs allowed by the scope for the client ID in use.

For a complete description of this process, check this video: https://youtu.be/SdGuTt98JRg.

Generic Client

Alternatively to Swagger UI, you can use whatever client software to test and check the APIs. See below.

Check system health with cURL

```
curl -X GET "http://localhost:8080/privacy-metrics/health" -H  "accept: */*"
```

Expected result:

```
"Everything's fine here! There are currently 9323 Privacy Metrics in the database at the moment"
```

Get the authorization token

To call Privacy Metrics APIs, you must obtain the credentials from the administrators of the authorization provider (KeyCloak in this case).

```
curl --location --request POST 'https://easypims.pimcity-h2020.eu/identity/auth/realms/pimcity/protocol/openid-connect/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=client_credentials' \
--data-urlencode 'client_id=<YOUR_CLIENT_ID>' \
--data-urlencode 'client_secret=<YOUR_CLIENT_SECRET>'
```

The response to this request will contain the access_token to use in all subsequent requests:

```
{"access_token":"<YOUR_ACCESS_TOKEN>"}
```

From now on, all requests to APIs will have to attach the access_token to be authorized.

Get the Privacy Metrics for the first 5 services in DB

```
curl -X GET "http://localhost:8080/privacy-metrics/privacy-metrics?limit=5&offset=0" \
-H  'accept: application/json' \
-H 'Authorization: Bearer <YOUR_ACCESS_TOKEN>'
```

Example result:

```
[
  [
    "alaska.edu",
    "3cb94130-a1eb-11eb-a48f-8c85904fb3aa"
  ],
  [
    "google.lk",
    "3caea27a-a1eb-11eb-a48f-8c85904fb3aa"
  ],
  [
    "jhu.edu",
    "3cad1d1a-a1eb-11eb-a48f-8c85904fb3aa"
  ],
  [
    "it.altervista.org",
```

```
    "3ca8dd0e-a1eb-11eb-a48f-8c85904fb3aa"
  ],
  [
    "ard.de",
    "3cad59e2-a1eb-11eb-a48f-8c85904fb3aa"
  ]
]
```


Changes

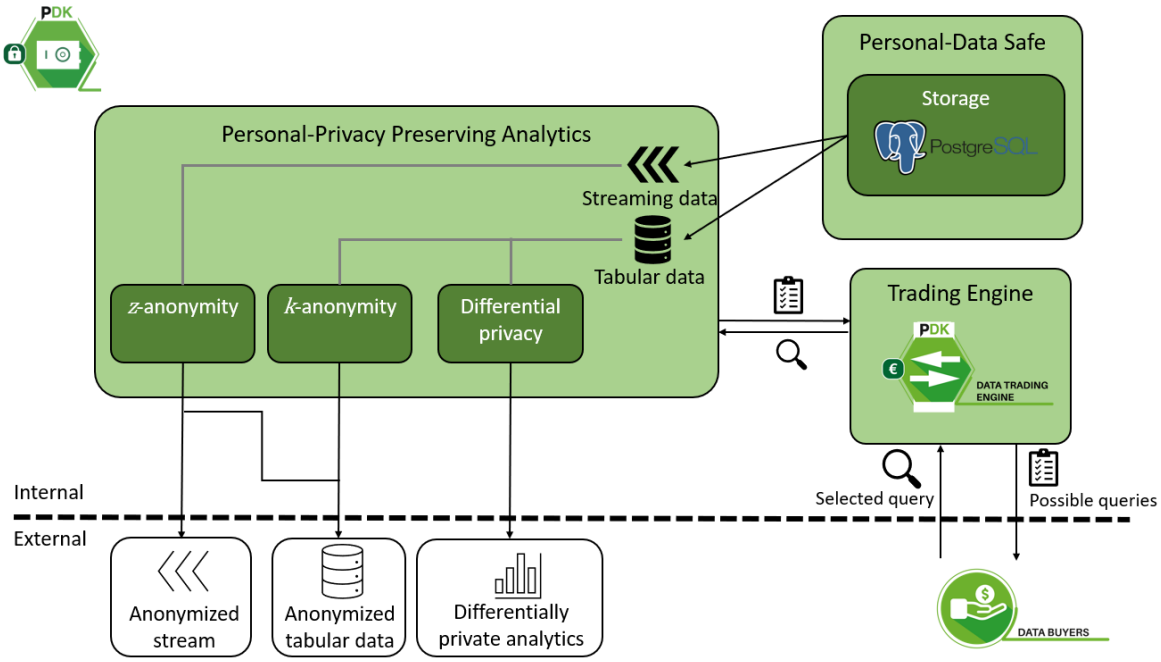
- 4 October 2021: added Auth authentication
- 16 November 2021: improved GET /privacy-metrics endpoint to include richer information

4.- Personal Privacy Preserving Analytics (P-PPA)

Overview

The Personal Privacy Preserving Analytics (P-PPA) module has the goal of allowing data analysts and stakeholders to extract useful information from the raw data while preserving the privacy of the users whose data is in the datasets. It leverages concepts like *Differential Privacy* and *K-Anonymity* so that data can be processed and shared while guaranteeing privacy for the users.

P-PPA includes a set of functionalities that allow perform data operations preserving the major privacy properties*: k-anonymity*, *z-anonymity*, *differential privacy*. P-PPA is capable to handle different sources of data inputs, that define which kind of privacy property is called into account: we have design solutions for tabular and batch stream, handled with **PostgreSQL**, **MongoDB**, and CSV modules, and live stream data. The figure below depicts the P-PPA architecture.



Installation

P-PPA is a Python3 module that requires the following libraries, which are also listed in the `requirements.txt` file:

```
aniso8601==9.0.1
certifi==2021.5.30
chardet==3.0.4
click==8.0.1
cycler==0.10.0
diffprivlib==0.3.0
Flask==1.1.2
Flask-RESTful==0.3.8
idna==2.10
```

```
itsdangerous==2.0.1
Jinja2==3.0.1
joblib==1.0.1
kiwisolver==1.3.1
MarkupSafe==2.0.1
matplotlib==3.3.3
numpy==1.18.5
pandas==1.1.5
Pillow==8.3.0
psycopg2==2.8.6
pymongo==3.11.2
pyparsing==2.4.7
python-dateutil==2.8.1
pytz==2021.1
PyYAML==5.4.1
requests==2.24.0
scikit-learn==0.24.2
scipy==1.7.0
six==1.16.0
SQLAlchemy==1.3.20
threadpoolctl==2.1.0
urllib3==1.25.11
Werkzeug==2.0.1
zanon==0.3.3
zanon==0.3.2
```

To install the module, it is only necessary to clone this repository and install the dependencies. The testing has been done on **a Linux Ubuntu 20.04** Machine, but the module is supposed to work on any Python installation.

Usage

The input data need to be in **pandas.Dataframe** format; in these examples, it is supposed to be already present. The chosen datasets for these examples is called "**Adult**", available in `data_manage/data_samples` folder. "**Credit**" and "**Diabetes**" datasets are also available in the same folder. Detailed API documentation is available in the `documentation/doc_sphinx folder` in the Sphinx format.

Following there are some usage examples:

1. K-anonymity, performing Mondrian algorithm.

Creating the Mondrian class with just the k parameter: all column attributes will be taken into account to perform the Mondrian algorithm.

```
from algorithms.kanonymity.mondrian.mondrian import Mondrian

mondrian = Mondrian(3)

k_anonymized_dataframe = mondrian.perform(input_dataframe)
```

2. K-anonymity, without performing Mondrian algorithm.

Differently from the previous example, here are selected the column indexes 2, 3, 4 and 5, corresponding *to fnlwgt, education, education-num* and *marital-status* attributes. Having specified these columns, the **Mondrian** algorithm will not be performed (obviously for this particular data and columns. For further details please check "*perform*" and "*_check_kanon_from_columns*" class methods documentation in the `mondrian.py` module).

```
from algorithms.kanonymity.mondrian.mondrian import Mondrian

mondrian = Mondrian(3, user_choice_index=[2,3,4,5])

k_anonymized_dataframe = mondrian.perform(input_dataframe)
```

3. K-anonymity, select QIs and whitelist columns.

In this example the parameters "*qi_indexes*" and "*whitelist*" are used to select respectively:

- which columns need to be considered as quasi-indetifiers, selected to achieve k-anonymity
- white-list columns, selected to be displayed among transformed attributes but conserved untouched as they are (they don't contribute to k-anonymity conversation: their use is related to performing anonymization on some attributes and conserving one o more possible labels (the white-listed attributes) to extract statistics and be able to train machine learning models).

Some more details:

- if "*qi_indexes*" isn't specified, all columns are considered as quasi-identifiers.
- if a column index in the "*whitelist*" list is also present in "*qi_indexes*" one, it's behavior will be overwritten and treated as white-listed.

Note: this to parameters have been introduced for an administrator use.

```
from algorithms.kanonymity.mondrian.mondrian import Mondrian

mondrian = Mondrian(3, qi_indexes=[1,2,4,6], whitelist=[8,9])

k_anonymized_dataframe = mondrian.perform(input_dataframe)
```

4. Differential privacy, performing the mean of the first and third columns.

For further details please check "*Dp_IBM_wrapper*" class documentation in the "*dp_IBM_wrapper.py*" module.

```
from algorithms.differential_privacy.dp_IBM.dp_IBM_wrapper import
Dp_IBM_wrapper

mean = Dp_IBM_wrapper([0,2], "mean", 0.6)

ret_mean = mean.perform(input_dataframe)
```

5. Differential privacy, performing the histogram on the first column.

For further details please check "*Dp_IBM_wrapper*" class documentation in the "*dp_IBM_wrapper.py*" module.

```
from algorithms.differential_privacy.dp_IBM.dp_IBM_wrapper import
Dp_IBM_wrapper

histogram = Dp_IBM_wrapper([0], "histogram", 0.6)

hist, bins = histogram.perform(input_dataframe, bins=6)
```

This is the same output that you would obtain exploiting numpy library. The following code it's just an example of a way with which you can use this output.

```
from matplotlib import pyplot as plt

width = 0.7 * (bins[1] - bins[0])

center = (bins[:-1] + bins[1:]) / 2

plt.bar(center, hist, width=width)

plt.show()
```

6. Differential privacy, performing the 2d histogram on the first and the third columns.

For further details please check "Dp_IBM_wrapper" class documentation in the "dp_IBM_wrapper.py" module. Please take in mind that the histogram from the first and the third column of "Adult" dataset has no semantic meaning.

```
from algorithms.differential_privacy.dp_IBM.dp_IBM_wrapper import
Dp_IBM_wrapper

histogram2d = Dp_IBM_wrapper([0,2], "histogram2d", 0.6)

matrix2d, xedge, yedge = histogram2d.perform(input_dataframe)
```

This is the same output that you would obtain exploiting **numpy** library.

7. Z-Anonymity in a Data Stream.

Z-anonymity is an anonymization property and algorithm for data streams. To use it, you need to have a Data Frame and indicates which columns identify the time, the users and the attributes.

```
import algorithms.zanon

import pandas as pd

z = algorithms.zanon.zanon(10, 3)
```

```
df = pd.read_csv("sample-stream.csv")

anon_df = z.perform(df, "time", "user", "item")
```

Web Service

The P-PPA implement a simpe Web server written in Flask that allow to use the P-PPA as a Web Service, allow any component (written in any language) to use the P-PPA, potentially hosted in a different server.

To start it, just run:

```
cd restapi

python init_flask.py
```

The Web API are documented in the OpenAPI format at: https://gitlab.com/pimcity/wp5/open-api/-/blob/master/WP2/privacy-preserving-analytics.yml.

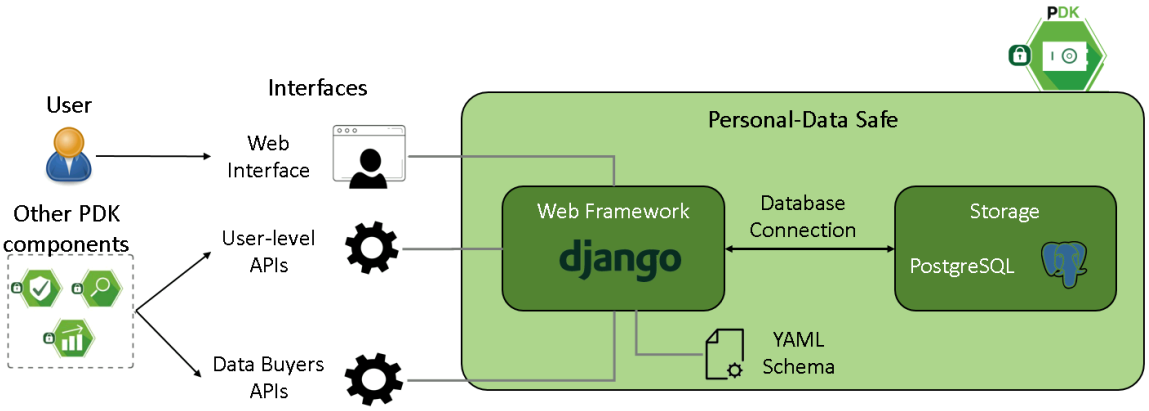
Changes

No Changes

5.- Personal Data Safe (P-DS)

Overview

The Personal Data Safe (P-DS) is the means to store personal data in a controlled form. It implements a secure repository for the user's personal information like navigation history, contacts, preferences, personal information, etc. It gives the possibility to handle them though REST-based APIs or a web interface. Thanks to the REST APIs, the P-DS can be accessed also by other components of the PDK. The architecture of the PDK is depicted in the figure below.



Installation

The documentation and the following instructions refer to a Linux environment (Ubuntu has been used for testing), with **Python 3.8.2** and **pip 20.0.2** installed. The P-DS project has been cloned from the GitLab repository at: https://gitlab.com/pimcity/wp2/personal-data-safe.

Follow accurately the next steps to quickly set-up the P-DS on your server. The package comes with a frontend and a backend already implemented, but if the needs calls only for a ready API you can cancel the folder `/frontend` and skip the relative steps. All relevant steps are designed for a Linux machine, perform the equivalent procedure with other environments. A Dockerized version is available online at https://hub.docker.com/r/martino90/personal-data-safe, so that it can be executed on any system supporting docker. In the following, we report instructions for running the PDS natively.

First Steps:

Update all packages:

```
sudo apt-get update

sudo apt-get upgrade

sudo apt-get dist-upgrade
```

Install pip/pip3:

```
sudo apt-get python3-pip
```

Import the project from the GIT repository:

```
git clone https://gitlab.com/pimcity/wp2/personal-data-safe.git
```

Backend Setup:

Now enter the backend folder:

```
cd backend/
```

Create a python virtual environment and activate it:

```
python3 -m venv venv source venv/bin/activate
```

Install the python-dev library and also add wheel:

```
sudo apt-get install libpq-dev python-dev
```

```
pip install wheel
```

Install all requirements:

```
pip install -r requirements.txt
```

Now we need to set-up the database, for this project the default one is PostegreSQL. If there is no need to change database, then the app will be ready to use after the next steps. Instead, if another database is needed, check the django documentation at https://docs.djangoproject.com/en/3.2/ref/databases to find out what steps to follow.

- First install PostegreSQL:

```
sudo apt-get update
```

```
sudo apt-get install python-pip python-dev libpq-dev postgresql
postgresql-contrib
```

- Create a database and database user. The default settings are:

```
db_name = pds_postgres
```

```
db_username = admin
```

```
user_password = admin_secret_password
```

- The steps to create the new database are the following:

```
sudo su - postgres
```

```
psql
```

```
CREATE DATABASE <db_name>;
```

```
CREATE USER <db_username> WITH PASSWORD '<user_password>';

ALTER ROLE <db_username> SET client_encoding TO 'utf8';

ALTER ROLE <db_username> SET default_transaction_isolation TO
'read committed';

ALTER ROLE <db_username> SET timezone TO 'UTC';

GRANT ALL PRIVILEGES ON DATABASE <db_name> TO <db_username>;

\q

exit
```

- It is now possible to use the database using the user and credentials registered:

```
psql --host <localhost or ip_addr> --port <port num> --username
<db_username> <db_name>
```

Connect Django to the PostgreSQL

```
pip install django psycopg2
```

If you don't want to use the default username and password, you will need to change them in the settings in the project file `backend/config/settings.py`.

```
DATABASES = {

      'default': {

      'ENGINE': 'django.db.backends.postgresql_psycopg2',

      'NAME': '<_db_name_>',

      'USER': '<db_username>',

      'PASSWORD': '_<user_password>_',

      'HOST': 'localhost',

      'PORT': '',

      }

}
```

Now Migrate the database. NB that it's better and safer to perform migrations for each app:

```
python manage.py makemigrations app_users

python manage.py makemigrations app_personal_data

python manage.py migrate
```

Create a Django SuperUser, which will also work as admin in the admin page:

```
python manage.py createsuperuser
```

It is now possible to run server and access admin:

```
python manage.py runserver 0.0.0.0:8000
```

```
admin found @ localhost:8000/admin
```

Frontend Setup:

This section focuses on the frontend setup of the application. The frontend is developed in javascript using the ReactJS framework. To begin, enter the frontend folder:

```
cd frontend/
```

First install NodeJs.

- Enable the NodeSource repository by running the following curl command as a user with sudo privileges:

```
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -
```

- Once the NodeSource repository is enabled, install Node.js and npm by running:

```
sudo apt install nodejs
```

- Verify that the Node.js and npm were successfully installed by printing their versions:

```
node --version
```

```
npm –version
```

Install dependencies for React:

```
cd Frontend
```

```
npm install
```

In case of some error like *"This version of npm is compatible with lockfileVersion@1, but package-lock.json was generated for lockfileVersion@2. I'll try to do my best with it!*" try running this command:

```
sudo rm -rf node_modules package-lock.json && npm install
```

Run React Server:

```
npm run
```

The project should now be working. Activate concurrently the Django and the ReactJS server and the platform should be fully functional.

Test Data:

If needed some scripts are already present for quickly creating some test data and check that the platform is correctly working.

They are `script_create_random_user.py`, `script_create_random_browsing_history.py` and `script_create_random_location_history.py`.

Random users:

To create random users use the script create_random_user.py. This script will generate a given amount of users, with a random username and a fixed password: test_password. Only need to keep track of the usernames created. Users will have some personal fields already randomly generated.

Usage:

```
python create_random_user.py <int:number_of_new_users>
```

Example of usage:

```
python create_random_user.py 1
```

Example of output:

```
{'username': 'isabella_wilson'}
```

```
{'id': 43280, 'value': {'birth-data': '1999-03-16'}, 'metadata':
'metadata-birth-data', 'data_type': 'birth-data', 'group_name':
'personal-details', 'description': 'description-birth-data',
'created': '2021-05-03T14:42:27.345054Z'}
```

Random Browsing History:

To generate random browsing history data for a specific user, use the script create_random_browsing_history.py. This script will generate a given amount of browsing history data, and assign them to a given user.

Usage:

```
python create_random_browsing_history.py <str:username>
<int:number_of_new_data>
```

Example of usage:

```
python create_random_browsing_history.py isabella_wilson 2
```

Example of output:

```
{'id': 43281, 'value': {'visited-url': {'url':
'http://www.msn.com', 'title': 'Msn', 'time': '2016-03-06
04:58:20'}}, 'metadata': 'metadata-visited-url', 'data_type':
'visited-url', 'group_name': 'browsing-history', 'description':
'description-visited-url', 'created': '2021-05-
03T14:48:16.934886Z'}
```

```
{'id': 43282, 'value': {'visited-url': {'url':
'http://www.onet.pl', 'title': 'Onet', 'time': '2018-08-29
03:49:59'}}, 'metadata': 'metadata-visited-url', 'data_type':
'visited-url', 'group_name': 'browsing-history', 'description':
'description-visited-url', 'created': '2021-05-
03T14:48:16.941765Z'}
```

Random Location History:

To generate random location history data for a specific user, use the script create_random_location_history.py. This script will generate a given amount of location history data, and assign them to a given user.

Usage:

```
python create_random_location_history.py <str:username>
<int:number_of_new_data>
```

Example of usage:

```
python create_random_location_history.py isabella_wilson 2
```

Example of output:

```
{'id': 43286, 'value': {'visited-location': {'latitude':
46.05635313711818, 'longitude': 12.831298674353643, 'time': '2010-
03-04 04:45:20', 'description': 'example'}}, 'metadata':
'metadata-visited-location', 'data_type': 'visited-location',
'group_name': 'location-history', 'description': 'description-
visited-location', 'created': '2021-05-03T14:49:59.668810Z'}

{'id': 43287, 'value': {'visited-location': {'latitude':
43.880077139481415, 'longitude': 15.797250794542236, 'time':
'2012-07-28 10:55:07', 'description': 'example'}}, 'metadata':
'metadata-visited-location', 'data_type': 'visited-location',
'group_name': 'location-history', 'description': 'description-
visited-location', 'created': '2021-05-03T14:49:59.675079Z'}
```

Usage

Data Model:

The data model of the application consists of 2 main classes:

- User class
- PersonalInformation class

The User class is used to store the information about the single user and it extends the *AbstractUser* class provided by Django. The class does not modify much the parent class, since it doesn't need to store additional information. Each user is linked to a set of personal information; the information about this relationship can be found in the *PersonalInformation* class.

On the other side, personal information are modeled by the PersonalData class that is used to store arbitrary types of data, e.g., user static information (name, surname, year of birth, email, etc.), browsing history, location history. The value field of the PersonalData class is defined as a JSONField object; in this way, user can store both elementary data, such as int, string, etc., and more structured data, such as dictionaries, that are represented as JSON objects. The PersonalData class has also user attribute that is defined as a ForeignKey object, so that each PersonalData instance has a reference to the user that owns the entry. This solution emulates what is done in a classic relational

system and it has been chosen because it proved to be the most efficient especially with big volumes of data. Others field of the PersonalData class provides additional information related such as:

- **group**_name: it's the semantic group the entry belongs to.
- **metadata**: it's the name that identifies a personal information inside a group
- **type**: it's the type of the information. Possible types are int, string, date, boolean, float, dict. Since each entry is modeled as a JSON object, elementary type information are stored in the form `{'type': value}`, while dict information in the form `{'subfield1': value1, 'subfield2', value2}`.

Schema:

The schema is stored in the data safe file system and loaded at application initialization, specific functions ensure that the schema defined follows some basics guidelines that will be defined later in this paragraph. The whole project logic has been developed prioritizing ease of usage: the main goal is to only change/modify the schema for the whole application (backend and frontend) to adapt to the change.

The schema defines the kind of information that the data safe can accept; each information is characterized by the following basic attributes in the schema:

- **group-name:** it defines the high level group that the data belongs to, such as *personal-details*, browsing-history, ect.
- **name**: it's the name that identifies the information inside a group, e.g. *birth-date* in the *personal-details* group
- **type:** it defines the type associated to the information, e.g. *birth-date* must be a Date

The schema is used to validate the input provided by the user, in order to control that the inserted data complies with the information that the P-DS can store. The schema can be updated if the user wants to store additional information: new types must be declared using the same name-type syntax used for old types, so that the application can handle changes in the schema.

The schema is used also to read data from the database: data that no longer match the schema are simply ignored, so that users can still have access to them (returning to an old version of the schema for example).

```
name: "PIMCity default PDS schema"
version: 0.2
author: "Federico Torta, Annaloro Enrico, Martino Trevisan"
content:
- group-name: personal-details (1)
  user-insertion: true (2)
  user-update: true (3)
  add-zip-file: false (4)
  extract-json: true (5)
  types: (6)
    - name: first-name
      type: string
    - name: last-name
      type: string
```

```
    - name: birth-data
      type: date
    - name: age
      type: int
- group-name: browsing-history (7)
  user-insertion: false
  user-update: false
  add-zip-file: true
  extract-json: true
  types:
    - name: visited-url
      historical: true (8)
      type: dict (9)
      fields: (10)
        - url: string
        - page-title: string
        - time: date
- group-name: location-history (11)
  user-insertion: false
  user-update: false
  visualization-hint: map (12)
  add-zip-file: true
  extract-json: true
  types:
    - name: visited-location
      historical: true
      type: dict
      fields:
        - latitude: float
        - longitude: float
        - description: string
        - time: date
```

Explanation:

- Each group-name field represents a possible semantic high-level group. In this case the first group accepted by the P-DS is **personal-details** (1). The possible data that are considered part of the personal-details group are defined in the types field (6) which lists all the variants of a personal-details entry. The types field is basically a list of name-type pairs where name is the common name of the entry, while type is the actual type of the information. Each data is represented with the name and the type in order to let the P-DS store any kind of information, without being tied to a particular elementary type. The user can store any data he want but the P-DS will cast the inserted data to control the type compliance. Moreover each group can presents additional settings:
- user-insertion (2): if this field is true, the user will be able to add the information manually. The frontend will show an **Add** button that allows the user to directly insert new data from the page. Schema and type compliance controls are executed.
- user-update (3): if true, the user can update a P-DS entry from the UI, using an **Edit** button that allows to change just the value of the stored data.
- add-zip-file (4): if true, the user can add new data from a zip file, which contains a JSON file with the information to be inserted. The zip file is uploaded using an **Upload ZIP file** button of the UI. This function is just a prototype and aims to show the potentiality of the P-DS: the data import feature will be presumably used by

automatic software systems that can create the correct JSON file, giving the possibility to import user data (even in big volumes) from other data stores.

- extract-json (5): if true, the user can click an **Extract data** button to download the stored data in JSON format, inside a zip file. As well as the data import functionality, also the data extraction can be performed both at global level (all the groups together) or at group-level (so download just the data related to a certain group).
- (8) and (11) means that the other possible groups that the P-DS supports are location-history information and browsing-history information.
- some information types can be stored multiple times, because they are linked to a particular timestamp, such visited urls or the visited locations. In this case they are enriched with the flag historical, so that the user can have multiple entries for that type.

- in order to let the user store more complex and structured data, the schema supports the dict type (10). A dict object is basically a JSON object with a set of key-value pair. Each entry of the dict is a piece of the complete information and is represented as a name-value pair just as the elementary entry of the P-DS. In this way the same controls can be applied recursively on the elements of a dict object. The single component of a dict are displayed in the fields key (11).
- The visualization-hint (12) is a special property used for visualization aids in the frontend. Some keywords are mandatory and additional set-up in the fields will be mandatory to support the feature. As of today the active features are:
    - map : allows to plot, in a google maps canvas, one or more points given its coordinates. For this reason, when this field is set, it must also be set up two mandatory fields (10) latitude and longitude, as shown in the example above. Failing to do so will prevent the application from starting.
- On the frontend, it has also been implemented a sorting logic to order items based on a specific field. To automate this we introduced a set of fields which support this feature, this means that, by using specific names for the fields, some special features (such as ordering) will be available. The fields are the following
    - `time: date` : ascending and descending sort based on date
    - `title: string`: ascending and descending sort based on alphabetical order

```
# This example can be sorted both by time and by title
- group-name: browsing-history
  user-insertion: false
  user-update: false
  add-zip-file: true
  extract-json: true
  types:
    - name: visited-url
      historical: true
      type: dict
      fields:
        - url: string
        - title: string # <-- here
        - time: date # <-- here
```

Web API:

The specifications of the P-DS Web API can be found in the OpenAPI format on GitLab, at: https://gitlab.com/pimcity/wp5/open-api/-/blob/master/WP2/personal-data-safe.yml. The API allow other components, on behalf of the user, to create, read, update and delete personal information.

Requests shall be authenticated via the `Authorization: Bearer` HTTP Header. The Token can refer to a local user of the PDS: in this case, the token can be generated via username and password using the above mentioned API. The token can also be a JWT generated using a configurable OAuth provider (parameters are in the file `/backend/config/settings.py`). In this case, the access token must have the four scopes: `read:pds create:pds update:pds delete:pds`.

Data Buyer API:

This section describes the functionalities and requirements for the correct operation of the databuyers API.

The databuyers section provides the users information to the different databuyers. Each request must be authenticated via a JWT access token obtained via an OAuth procedure (configurable in the file `/backend/config/settings.py`). The token must have the scope `databuyer:pds`.

Requests are made to the endpoint `/data-buyers/get-data/` using the POST method, and the payload must contain the indication of the user and type of personal information to retrieve. Details and examples are available in the OpenAPI documentation available at: https://gitlab.com/pimcity/wp5/open-api/-/blob/master/WP2/personal-data-safe.yml.

Changes

- 29 November 2021: Added filtering by time in PDS API
- 16 December 2021: Refactored Data Buyer API
- January 15 2022: Added batch insert API

6.- Conclusions

This deliverable presented the User's guide of the following PIMCity PDK modules:

- **Personal Consent Manager (P-CM)**, from Task 2.1, the means to define once and for all the user's privacy preferences for consent management.
- **Personal Privacy Metrics (P-PM)**, from Task 2.2, easy to understand novel privacy metrics.
- **Personal Privacy Preserving Analytics (P-PPA),** from Task 2.3, controlling which data users are exposing
- **Personal Data Safe (P-DS),** from Task 2.4, the means to store personal data in a controlled form.

Along with guide, we release the final implementations of these modules, that are available online in the PIMCity GitLab repository. The current modules are being integrated in the deployment initiatives of the PIMCity project and might be further updated if needed.