5/30/2024

GDPR

General Data Protection Regulations

Hristova, Gabriela G.G.
FONTYS ICT

Table of Contents

Introduction to GDPR

The General Data Protection Regulation (GDPR) is a comprehensive European Union law that governs how organizations collect, process, and protect the personal data of individuals residing in the EU.

Here are some of the purposes of GDPR:

1. **Protect Fundamental Rights and Freedoms**: GDPR is designed to safeguard individuals' rights and freedoms, particularly their right to data protection and privacy. It aims to give individuals more control over their personal data and how it is used. [1]
2. **Harmonize Data Protection Laws**: By standardizing data protection regulations across the EU, GDPR ensures consistent rules for organizations operating within the region. This harmonization facilitates the free movement of personal data within the EU. [1]
3. **Modernize and Strengthen Data Protection**: GDPR updates and reinforces data protection principles to align with the realities of today's digital landscape, addressing modern data processing challenges and technologies. [1]

Principles of GDPR

Key Principles

Lawfulness, Fairness and Transparency

Personal data must be processed lawfully, fairly and in a transparent manner in relation to the data subject. [2]

Purpose Limitation

Personal data must be collected for specified, explicit and legitimate purposes and not further processed in a manner incompatible with those purposes. [2]

Data Minimization

Personal data collected must be adequate, relevant, and limited to what is necessary for the stated purposes. [2]

Accuracy

Personal data must be accurate and kept up to date. [2]

Storage Limitation

Personal data shall be kept in a form which permits identification of data subjects for no longer than is necessary for the purpose for which it is processed. [2]

Integrity and confidentiality

Personal data must be processed in a manner that ensures appropriate security, including protection against unauthorized or unlawful processing and accidental loss, destruction, or damage. [2]

Accountability

The controller shall be responsible for and be able to demonstrate compliance with the GDPR principles. [2]

Data Subject Rights & Key Requirements

Here are the Data Subject Rights:

1. Right to be informed about data processing activities
2. Right of access to personal data
3. Right to rectification of inaccurate personal data
4. Right to erasure/right to be forgotten
5. Right to restrict processing
6. Right to data portability
7. Right to object to processing
8. Rights related to automated decision-making including profiling

Here are some of the key requirements:

1. Lawful basis for processing personal data
2. Consent requirements for data processing
3. Data protection by design and default
4. Records of processing activities
5. Data protection impact assessments (DPIAs)
6. Data protection officers
7. Data breach notification
8. Cross-border data transfers

GDPR gives data subjects specific rights like access to their data, data portability, right to be forgotten, and rights around automated decision-making. It mandates organizations to implement data protection by design and default. [3]

Data Complexities

Distributed data processing models

Considering the rapidly evolving nature of data and its varied applications, the landscape is witnessing a surge in the development of data tools and design strategies to accommodate these changes. Here are the five models for integrating data into the development process.

Data-centric architecture development

Data-centric architecture is an approach where data is treated as the primary and permanent asset, driving the design, functionality, and decision-making processes of an organization. Key points for developing a data-centric architecture include:

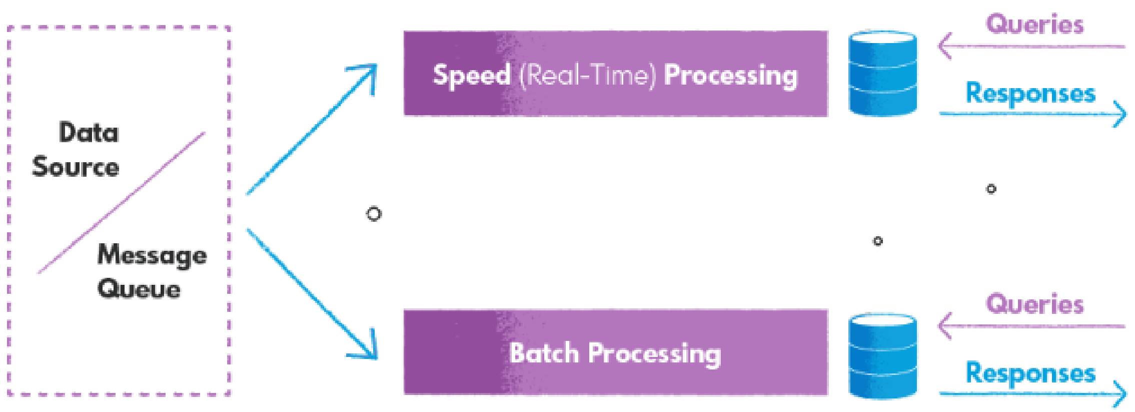
1. Defining Business Objectives [4]
    a. Define goals that the data architecture needs to support
    b. Determine who needs access to what data and how they will use it
    c. Establish data standards to ensure data quality, security, and compliance
2. Data Modelling and Integration [4]
    a. Create a model, that defines structures, relationships, and assassinations between different data entities
    b. Use data integration tools and processes to make easier transfer and synchronization of data across systems
    c. Adopt big data management tools, like Apache, Kafka, or Spark for efficient data integration and processing
3. Separation of Concern [4]
    a. Separating responsibilities of data storage can create more reliable, efficient, and scalable systems
4. Scalability and Performance [4]
    a. Optimize to efficiently process massive data sets (big data)
    b. Ensure the architecture can scale up and down to accommodate changes in data workload
5. Security and Privacy [4]
    a. Implement security measures and access control to protect sensitive data
    b. Make sure it is in line with the privacy regulations (ex. GDPR)
6. Adaptability and Agility [4]
    a. Implement design to accommodate rapidly expanding data workloads

By adopting a data-centric architecture, organizations can streamline operations, reduce redundant data, enable better decision-making through improved analytics.

There are two well-known data-centric architectures: Lambda and Kappa.

Lambda Architecture

Lambda Architecture is a data processing architecture that aims to provide a scalable, fault-tolerant, and flexible system for processing big data. One critical feature of this architecture is that it uses two separate data processing systems to handle several types of data. First system (batch processing system), processes data in large batches and stores the results in centralized data store. Second system (stream processing system), processes data in real-time as it arrives and stores the results in a distributed data store. [5]



Real-time stream processing and batch processing in Lambda Architecture

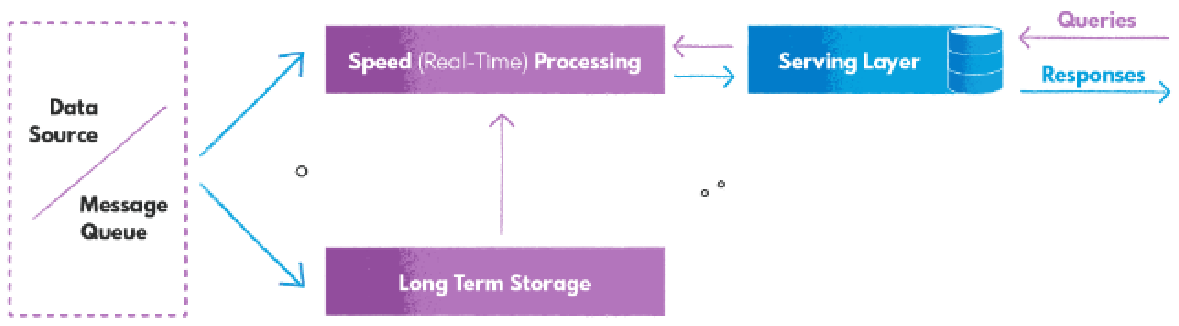
The Lambda architecture has four main layers: Data Ingestion Layer, Batch Layer, Speed Layer, and Serving Layer.

The Lambda Architecture offers several advantages, including scalability, fault tolerance, and flexibility. It is designed to manage large volumes of data and supports horizontal scaling. The architecture's multiple layers ensure reliable data processing and storage. Additionally, its flexibility allows it to handle a wide range of data processing workloads.

However, the Lambda Architecture also has some disadvantages. One of the main drawbacks is its complexity; setting it up and maintaining it can be challenging due to its multiple layers. Another issue is the potential for errors and discrepancies; with duplicated implementations of different workflows, inconsistencies can arise between batch and stream processing results. Lastly, reorganizing or migrating existing data within the Lambda Architecture can be quite difficult. [5]

Kappa Architecture

Kappa Architecture is an architecture developed as an alternative to the Lambda architecture, and instead of two processing systems, Kappa has only one that handles both batch processing and stream processing workloads, as it treats everything as stream. This allows it to provide more streamlined and simplified data processing pipeline while still providing fast and reliable access to query results. [5]



Real-time stream processing in Kappa Architecture

Compare to the Lambda architecture, Kappa has only one layer, which is the Speed Layer (Stream Layer).

The Kappa Architecture offers several advantages, including simplicity, a streamlined pipeline, ease of migrations and reorganizations, and tiered storage. With only one processing system, it is simpler to set up and maintain compared to the Lambda Architecture. Additionally, performing migrations and reorganizations is easier, as new data streams can be created from the authorized data store. [5]

However, there are still some disadvantages. Although the Kappa Architecture is simpler than Lambda, it can still be complex to implement. Another potential drawback is the costly infrastructure and scalability issues. To mitigate costs, using a data lake approach from a cloud provider can be a more cost-efficient solution. [5]

Lambda vs Kappa

In terms of data processing systems, the Lambda Architecture has two separate systems to handle several types of data workloads. This dual-system approach requires learning and maintaining two processing frameworks and supporting daily code changes in both. In contrast, the Kappa Architecture uses a single system to manage all data processing, eliminating the need for maintaining separate codebases for batch and stream processing. [5]

Regarding data storage, the Lambda Architecture features a distinct long-term data storage layer, used for storing historical data and performing complex aggregations. The Kappa Architecture, on the other hand, does not have a separate long-term storage layer; all data is processed and stored through the streaming system. [5]

The Lambda Architecture is more complex to set up and maintain due to its dual data processing systems. While the Kappa Architecture is simpler in this regard, it requires thinking of all data as streams and experience in stream processing and distributed systems. [5]

Data stream processing

Stream processing is acting on a series of data at the moment it is created. This often includes performing multiple tasks on the incoming data, which can be executed serially, in parallel, or both. This workflow is known as a stream processing pipeline. It encloses the generation of streaming data, its processing, and its delivery to a destination. Actions in stream processing include aggregations, analytics, transformations, enrichment, and ingestion. [6]

Data event sourcing

The fundamental idea of event sourcing is to ensure that every change to the state of an application is captured in an event object. These event objects are stored in the sequence they were applied and maintained for the same duration as the application's state. [7]

Advantages

1. Events are immutable and can be stored using an append-only operation. Recording events for handling at the appropriate time can simplify implementation and management.
2. This approach helps prevent conflicts from concurrent updates by avoiding direct updates to objects in the data store.
3. Provides a complete audit of all state changes, allowing for temporal queries and data analysis over time.
4. Event sourcing supports event-based integration with external systems by publishing events as they occur.
5. Enables replaying events to reconstruct the current state or create projections for specific use case

Issues & Considerations

1. Event store is the permanent source of information, the event data should never be updated
2. Adding timestamps can help avoid issues
3. Attempting two actions to add events for the same entity as the same time, can lead to the event store rejecting an even matching an existing entity identifier and event identifier
4. There is no standard approach, or existing mechanisms such as SQL queries, for reading the events to obtain information
5. The length of each event stream affects managing and updating the system
6. Even though event sourcing minimizes the chance of conflicting updates to the data, the application must still be able to deal with inconsistencies that result from eventual consistency and the lack of transactions

Distributed data structures (DDS)

A distributed data structure (DDS) is a self-managing storage layer designed to operate on a cluster of workstations. It provides a simple, conventional data structure interface to service authors while handling complex mechanisms like data partitioning, replication, scaling, and recovery behind the scenes

DDS presents a familiar data structure interface to developers, abstracting away the complexities of distributed systems. It offers high throughput, high concurrency, availability, incremental scalability, and strict data consistency through replication and coherence protocols. However, DDS does not support transactions across multiple elements or operations.

Compared to databases, a DDS sacrifices data independence for simplicity, offering a less flexible but higher-level abstraction than file systems. DDS provides better performance than databases for internet services by avoiding overheads like query parsing, though it lacks the full flexibility of SQL.

Here are some examples of DDS:

- RDD
- Alluxio
- Apache Ignite
- Redis

RDD (Resilient Distributed Datasets)

Resilient Distributed Data (RDD) is a fundamental data structure of Spark. Each dataset in an RDD is divided into logical partitions that can be computed on different nodes of a cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

RDD is a read-only, partitioned collection of records. RDDs can be created through operations on data from stable storage or other RDDs. They are fault-tolerant collections of elements that can be operated on in parallel

Alluxio

Alluxio is an open-source virtual distributed file system that serves as a data orchestration layer between computation frameworks and storage systems.

Alluxio provides a data abstraction layer that allows applications to access data from multiple storage systems through a unified interface. [9]

Apache Ignite

Apache Ignite is an in-memory computing platform that includes an in-memory data grid, in-memory database, support for streaming analytics, and a continuous learning framework for machine and deep learning. It provides in-memory speed and unlimited horizontal scalability. [8]

Redis

Redis is an open-source, in-memory data structure store that is widely used as a database, cache, and message broker.

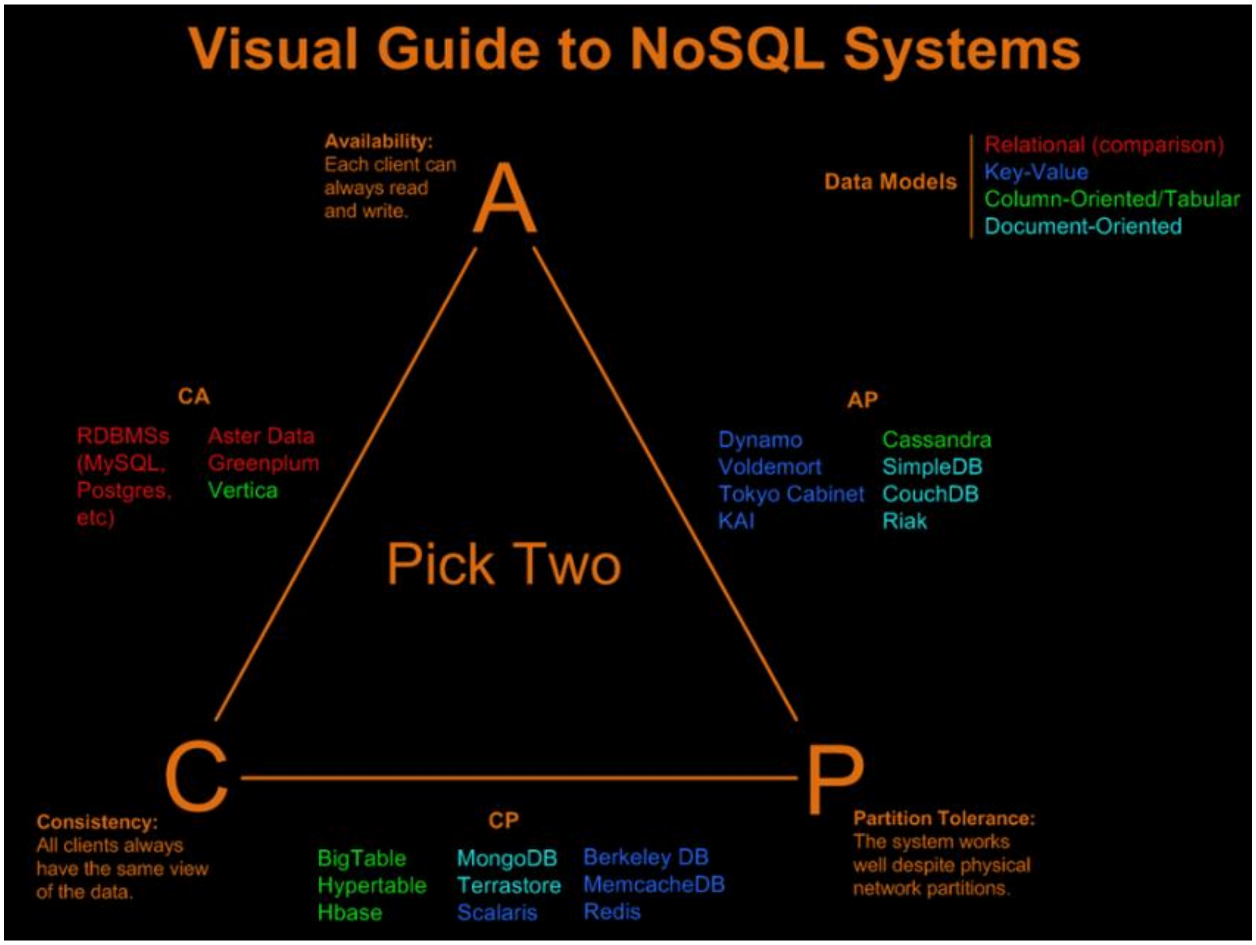
It is a NoSQL, key-value store that stores data in-memory, providing extremely fast read and write performance. It supports a variety of data structures like strings, hashes, lists, sets, sorted sets, bitmaps, hyperlogslogs, and streams, enabling it to handle diverse data models efficiency. Redis provides hight quality features like Redis Sentinel and Redis Cluster. [11][12]

Blockchain

Blockchain is a distributed data structure that enables secure and transparent storage across a decentralized network. It consists of a chain of blocks, where each block contains a set of transactions and is linked to the previous block by using cryptography, making immutable ledger of all transactions. Key components of a block are Header (identifies the block in the entire blockchain), Previous Bock Address (connects the i+1th block to the ith block using hash), Timestamp (time or date of creation for digital documents), Nonce, and Merkel Root (type of data structure frame). [13][14]

CAP Theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental principle in the design of distributed systems. It states that any distributed data store can provide only two of the following three guarantees: Consistency, Availability, and Partition tolerance.


Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.

GDPR Implementation

Data Mapping

In this group project, we have implemented several microservices, each with its own set of entities. The microservices currently available are:

User Microservice

The User microservice manages user information with an entity called User. This entity stores the following data:

- ID (sensitive)
- First name
- Last name
- Email (sensitive)
- Password (sensitive)
- Salt (sensitive)
- Token

Sensitive data includes the ID, email, password, and salt, which are not publicly displayed. The remaining information is publicly accessible.

Salt is a random string added to a password before hashing to defend against dictionary attacks and precomputed rainbow table attacks [15]. To protect passwords, we hashed it with a salt before storage. Both the hashed password and salt are stored in the database. When a user logs in, the entered password is hashed again using the stored salt for comparison with the stored hashed password. The email is only used during the login process and is not displayed in the application.

Tournament Microservice

The Tournament microservice manages tournament data with an entity called Tournament. This entity stores:

- ID (sensitive)
- Name
- Date
- Min/Max number of players
- Type
- Description
- Created timestamp
- Updated timestamp (sensitive to admins only)

Sensitive data includes the ID and updated timestamp, which are only visible to admins. All other information is publicly accessible.

Challenge Microservice

The Challenge microservice includes three entities: Challenge, ChallengeUser, and MatchMaking.

Challenge entity stores:

- ID (sensitive)
- Name
- Start date
- End date
- List of user IDs
- List of challenge IDs

ChallengeUser entity stores:

- ID (sensitive)
- User_ID (sensitive)
- Score
- Wins
- Ties
- Losses

MatchMaking entity stores:

- ID (sensitive)
- Challenger_ID (sensitive)
- Challenged_ID (sensitive)
- Score
- Status
- Game code (sensitive, not stored, provided by an external API)

Sensitive data in these entities includes IDs, user_IDs, challenger_ID, challenged_ID, and the game code. This sensitive data is not displayed to normal users, and the game code is provided from an external API, and it is not stored in our database.

In summary, our project consists of microservices designed with careful consideration of data sensitivity to ensure appropriate access and security.

Third-Party Management

As mentioned, we will have three microservices: User, Tournament, and Challenge. Since each microservice serves a different purpose, we have decided to use separate instances of the database, MySQL.

After discussions with the client, we opt for the need for a centralized location to store these instances. Due to that, we created an Azure account to host the MySQL instances. Setting up these instances on Azure will enable us to access, test, and deploy them efficiently. Hosting the three instances on Azure will also facilitate our preparation for the event at the beginning of June.

References

[1] - *Robinson, S., Castagna, R., & Lavery, T. (2024, March 27). General Data Protection Regulation (GDPR). WhatIs. https://www.techtarget.com/whatis/definition/General-Data-Protection-Regulation-GDPR*

[2] - *Ken.Bernadini. (2022, November 9). GDPR Requirements - Quick Guide on Principles & Rights. GDPR EU. https://www.gdpreu.org/gdpr-requirements/*

[3] - *M. (2024, May 7). GDPR Compliance Checklist: 10 Key steps (With infographic). CookieYes. https://www.cookieyes.com/blog/gdpr-checklist-for-websites/*

[4] - *Éclosion, D. (2023, July 28). Importance and implementation of Data-Centric Architectures in a digital economy. Data Éclosion. https://www.data-eclosion.com/en/data-centric-architectures/*

[5] - *Dorota-Owczarek. (2023b, April 24). Lambda vs. Kappa Architecture. A Guide to Choosing the Right Data Processing Architecture for Your Needs. nexocode. https://nexocode.com/blog/posts/lambda-vs-kappa-architecture/#data-processing-architectures*

[6] - *Hazelcast. (2024, January 12). What is stream processing? A layman's overview | Hazelcast. https://hazelcast.com/glossary/stream-processing/*

[7] - *RobBagby. (n.d.). Event Sourcing pattern - Azure Architecture Center. Microsoft Learn. https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing*

[8] - *Introduction to Apache Ignite. (n.d.). GridGain Systems. https://www.gridgain.com/resources/papers/introducing-apache-ignite*

[9] - *Alluxio. (n.d.). GitHub - Alluxio/alluxio: Alluxio, data orchestration for analytics and machine learning in the cloud. GitHub. https://github.com/Alluxio/alluxio*

[10] - *Sandesh. (2023, April 3). A quick overview of distributed Data Structures (DDS). Medium.* *https://medium.com/@s7326731/a-quick-overview-of-distributed-data-structures-dds-f741bf7b0ab7#:~:text=A%20distributed%20data%20structure%20(DDS,a%20self%2Dmanaging%20storage%20layer*.

[11] - *What is Redis Explained? | IBM. (n.d.).* *https://www.ibm.com/topics/redis*

[12] - *About - Redis. (2024, April 9). Redis.* *https://redis.io/about/*

[13] - *GeeksforGeeks. (2022, November 16). Blockchain Structure. GeeksforGeeks.* *https://www.geeksforgeeks.org/blockchain-structure/*

[14] - *Metaschool. (2023, January 3). What's a blockchain data structure like? Metaschool.* *https://metaschool.so/articles/blockchain-data-structure/*

[15] - *Wikipedia contributors. (2024, May 22). Salt (cryptography). Wikipedia.* *https://en.wikipedia.org/wiki/Salt_%28cryptography%29*