





Java Academy 2022



Database Interaction and ORM

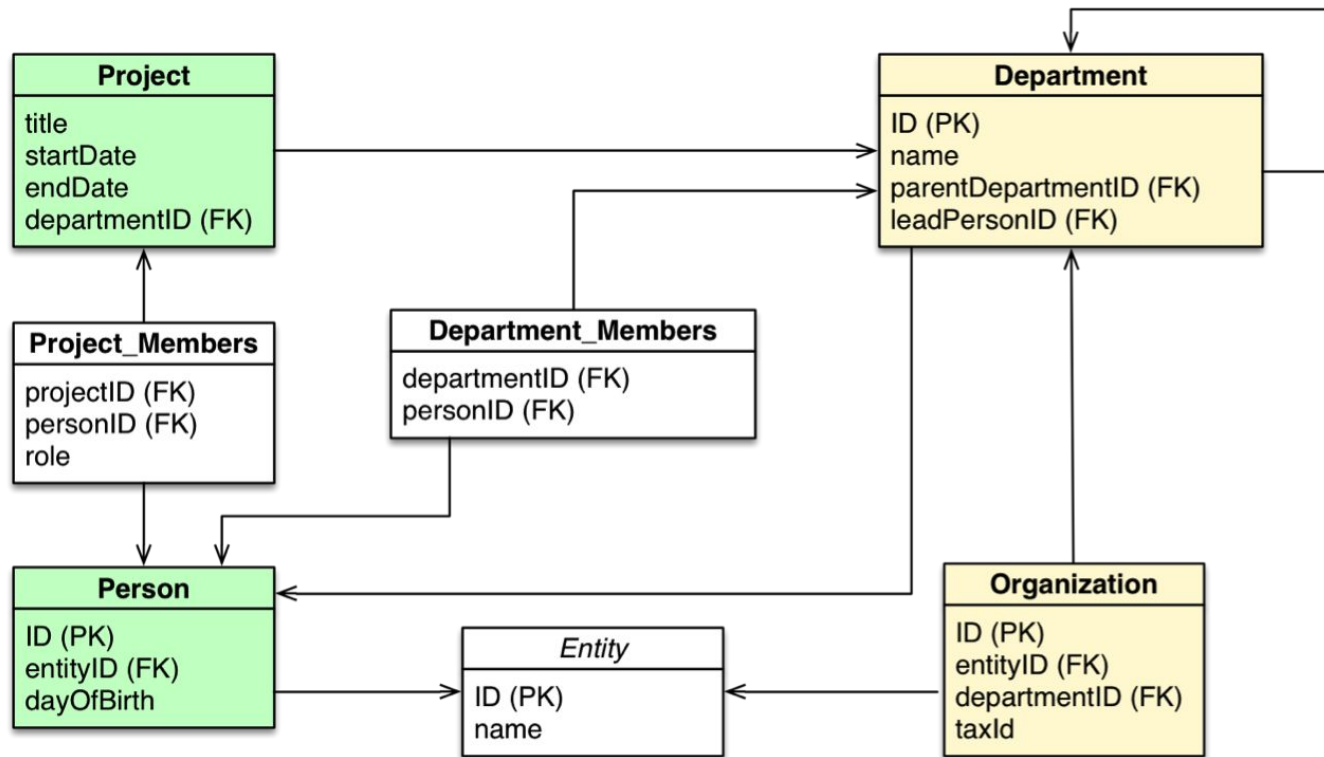


Relational Databases

- They are usually ACID.
- Based on the relational model of data. This model organizes data into one or more tables (or "relations") of columns and rows, with a unique key identifying each row.
- Rows are also called records or tuples. Columns are also called attributes. Generally, each table/relation represents one "entity type" (such as customer or product).
- The rows represent instances of that type of entity (such as "Place" or "chair") and the columns representing values attributed to that instance (such as address or price)



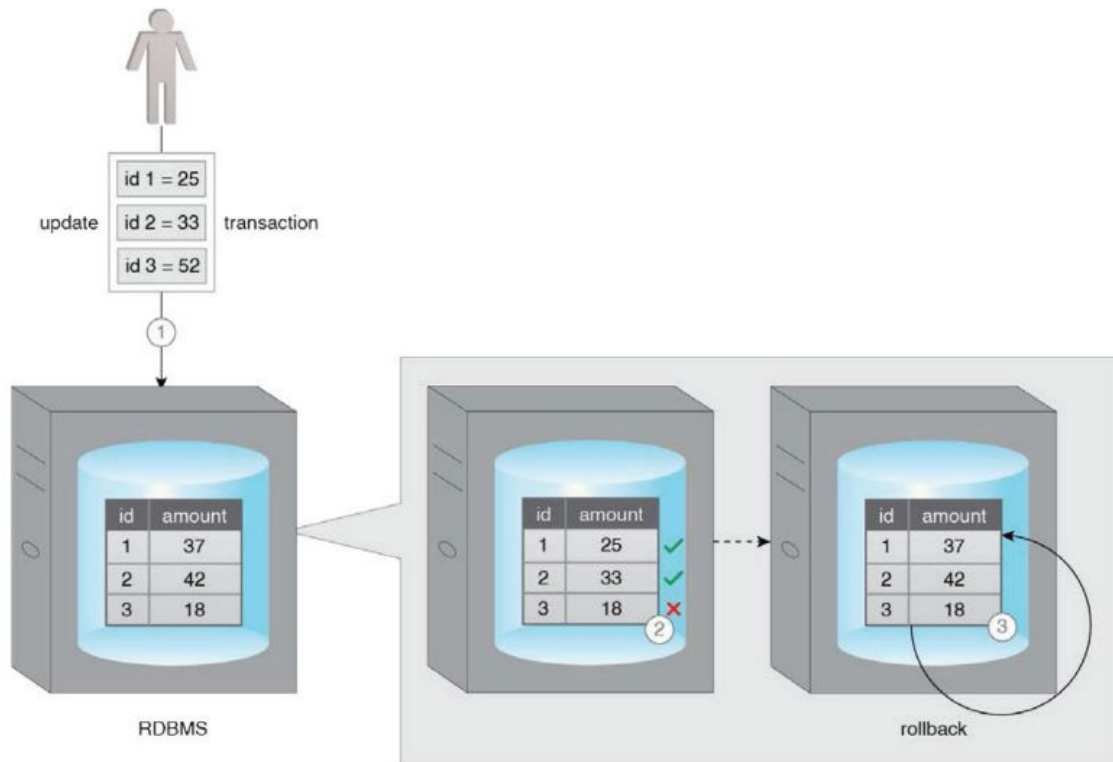
Relational Model





ACID: Atomicity

Ensures that all operations will always succeed or fail completely. In other words, there are no partial transactions.

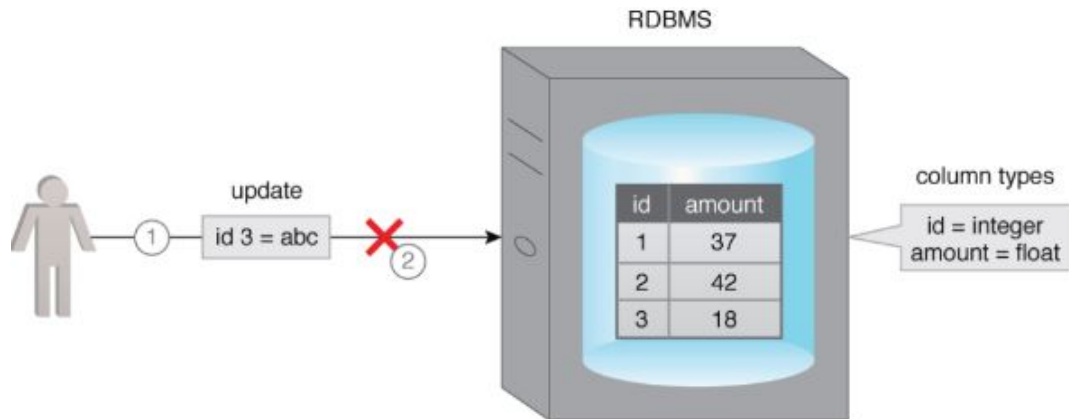




ACID: Consistency

Ensures that the database will only allow valid data, and will always be in a consistent state after an operation.

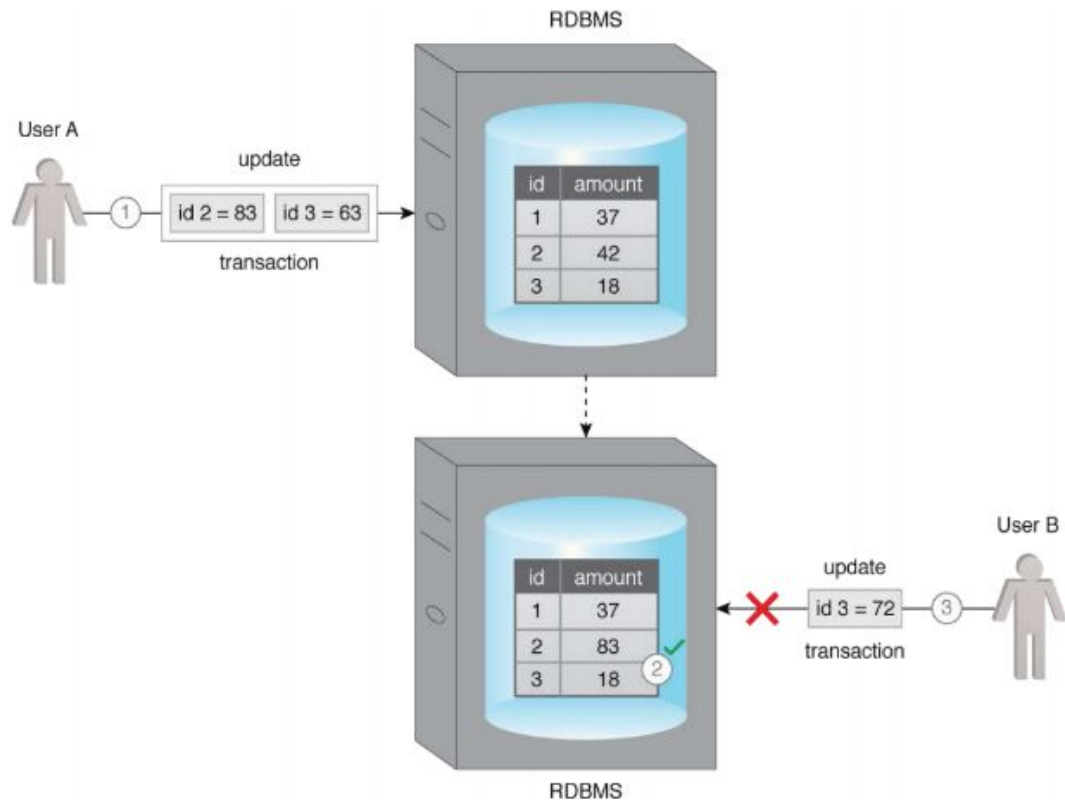
Any write followed by an immediate read is guaranteed to be consistent across multiple clients.





ACID: Isolation

Ensures that the results of a transaction are not visible to other operations until it is complete.

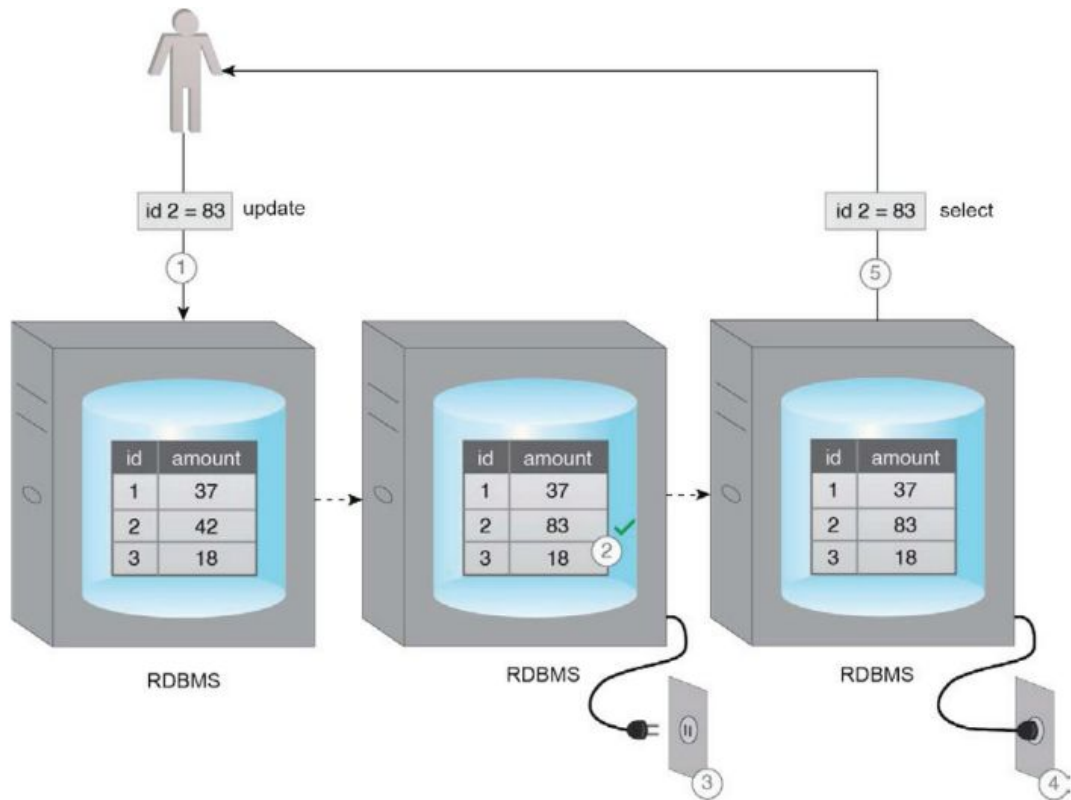




ACID: Durability

Ensures that the results of an operation are permanent.

In other words, once a transaction has been committed, it cannot be rolled back. This is irrespective of any system failure.





Relational Database Examples

- Oracle
- MySQL
- Postgres
- MSSQLServer
- H2



Non Relational Databases

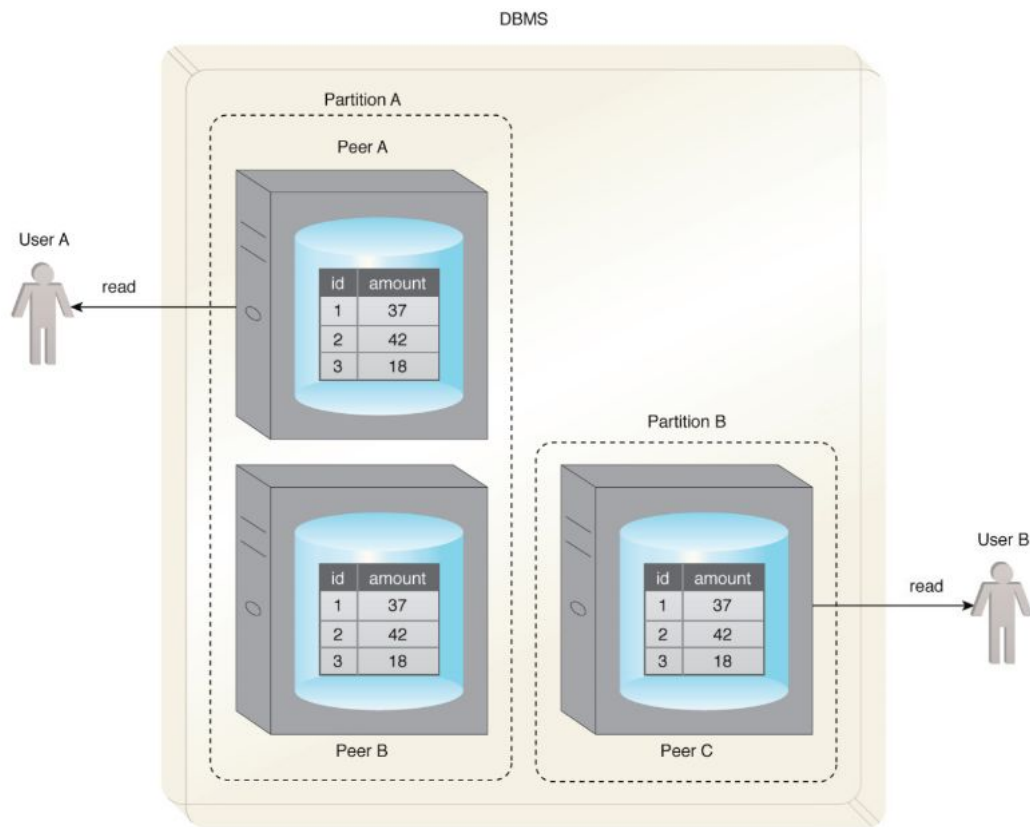
- They are usually BASE.
- BASE is a database design principle followed by database systems that leverage distributed technology.
- A non-relational database is a database that does not use the tabular schema of rows and columns found in most traditional database systems.
- Instead, non-relational databases use a storage model that is optimized for the specific requirements of the type of data being stored.
- For example, data may be stored as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.



BASE: Basically Available

Means that the database will always acknowledge the client's request either in the form of requested data or a success/failure notification.

Both users are serviced even though the database system has been partitioned as a result of a network failure.



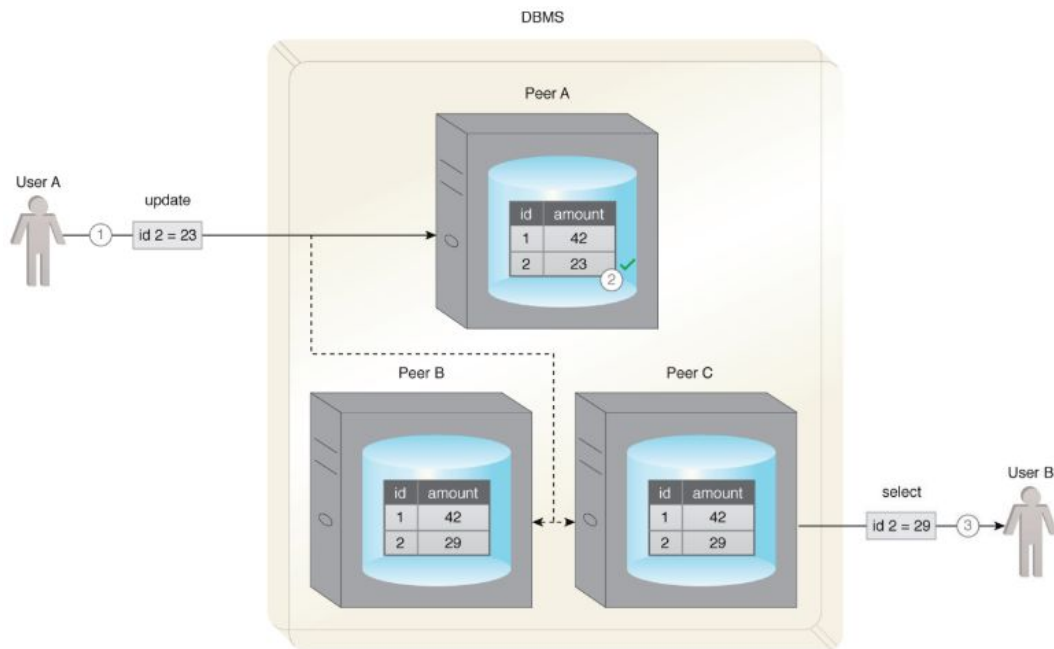


BASE: Soft State

Entails that the database may not be in a consistent state when data is read, thus the results may change if the same data is requested again.

This is because the data could be updated for consistency, even though no user has written to the data between the two reads.

This property is closely related to eventual consistency.



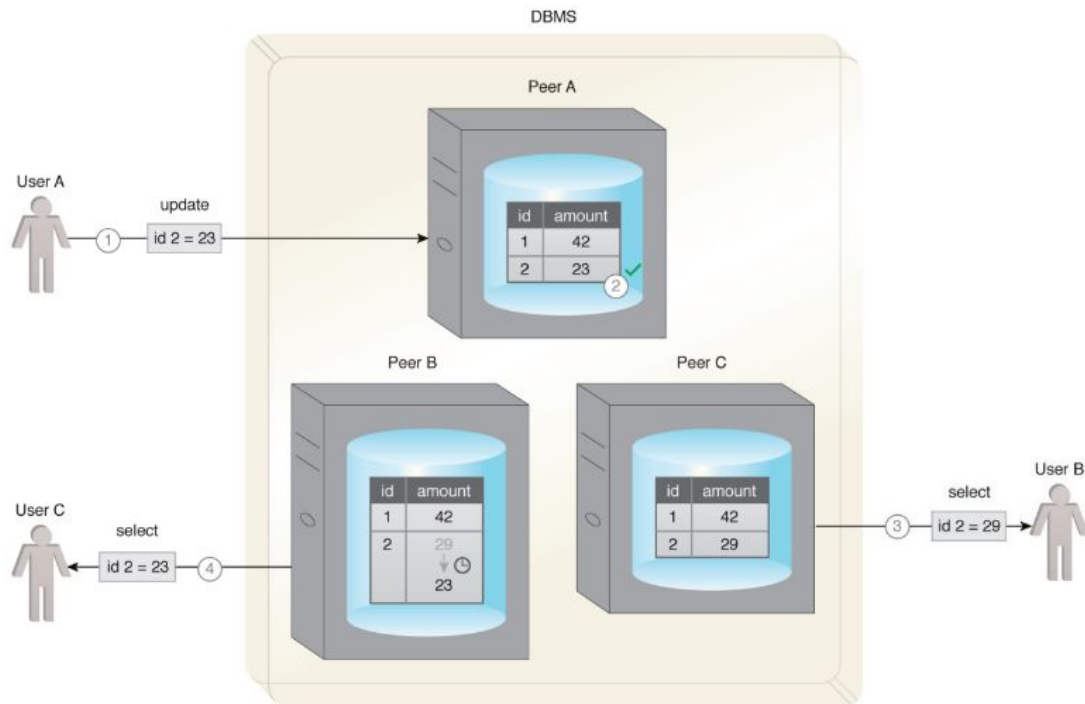


BASE: Eventual Consistency

The state in which reads by different clients, right after a write, may not return consistent results.

The database only attains consistency once the changes have been propagated to all nodes.

Therefore, while the database is in the process of attaining the state of eventual consistency, it will be in a soft state.





BASE vs ACID

- BASE emphasizes availability over immediate consistency, in contrast to ACID which ensures immediate consistency at the expense of availability due to record locking.
- This soft approach towards consistency allows BASE compliant databases to serve multiple clients without any latency albeit serving inconsistent results.
- However, BASE compliant databases are generally not used for transactional systems where lack of consistency can become a concern.



Types: Key-Value

Key-value storage devices store data as key-value pairs and act like hash tables.

The table is a list of values where each value is identified by a key.

The value is opaque to the database, and is essentially stored as a BLOB.

The value stored can be any aggregate, ranging from sensor data to videos.

key	value	
631	John Smith, 10.0.30.25, Good customer service	← text
365	1010110101011010101110101101010101010110101110	← image
198	<CustomerId>32195</CustomerId><Total>43.25</Total>	← XML

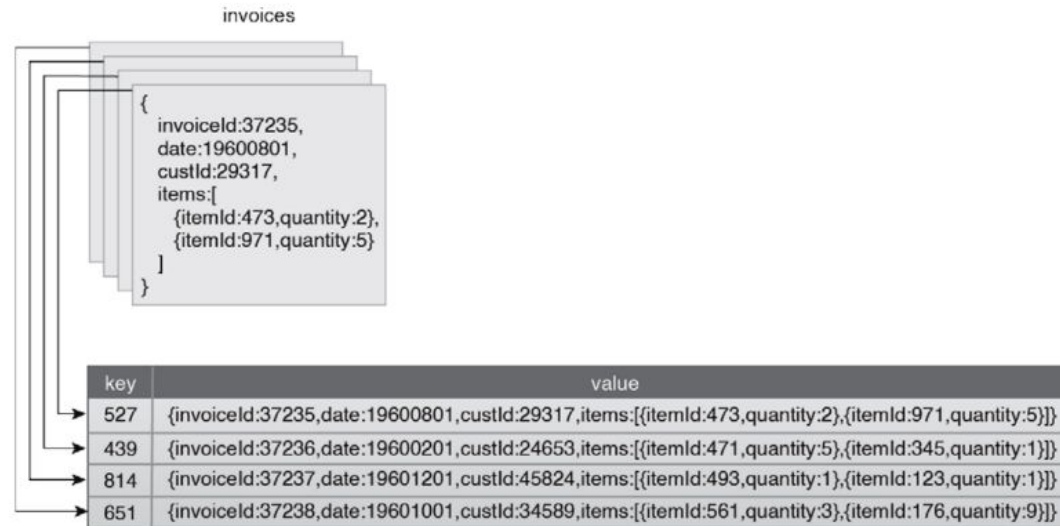


Types: Document

Document storage devices also store data as key-value pairs.

However, unlike key-value storage devices, the stored value is a document that can have a complex nested structure, such as an invoice.

The document can be encoded using either a text-based encoding scheme, such as XML or JSON, or using a binary encoding scheme, such as BSON (binary JSON).

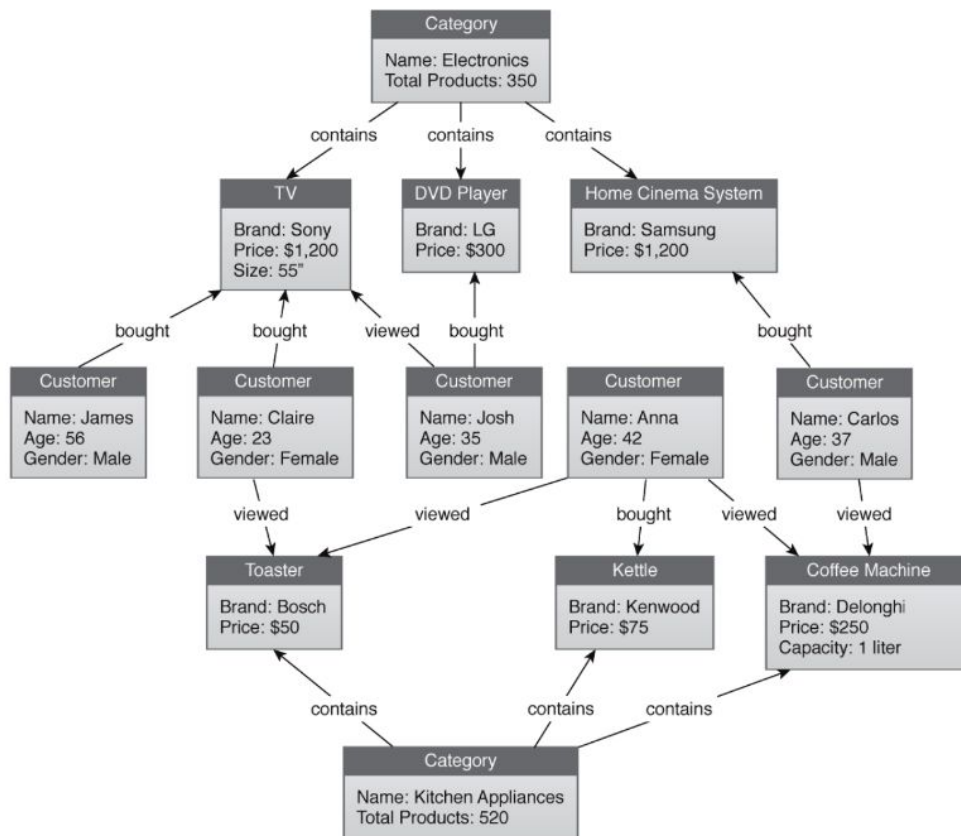




Types: Graph

Graph storage devices are used to persist inter-connected entities.

Unlike other NoSQL storage devices, where the emphasis is on the structure of the entities, graph storage devices place emphasis on storing the linkages between entities





Non Relational Database Examples

- Mongo
- Cassandra
- DynamoDB



Spring Data

Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store. It has a lot of features:

- Powerful repository and custom object-mapping abstractions
- Dynamic query derivation from repository method names
- Implementation domain base classes providing basic properties
- Support for transparent auditing (created, last changed)
- Possibility to integrate custom repository code





Spring Data JPA - Configuration

```
@SpringBootApplication  
@EnableJpaRepositories  
public class SpringDataJpaApplication
```

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'org.springframework.boot:spring-boot-starter-data-rest'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```



Spring Data JPA - Configuration

```
@SpringBootApplication
@EnableJpaRepositories (basePackages = "com.globant.persistence.repository")
public class SpringDataJpaApplication{
    ...
}
```

```
<jpa:repositories base-package="com.globant.persistence.repository" />
```



Spring Data JPA - Repository

The central interface in the Spring Data repository abstraction is Repository. It takes the domain class to manage as well as the ID type of the domain class as type arguments

```
@Indexed
public interface Repository<T, ID> {
    ...
}
```

```
public interface PersonRepository extends Repository<Person, Integer> {
    ...
}
```




Spring Data JPA - Main Interfaces

CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed.

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
  
    <S extends T> S save(S entity);  
  
    Optional<T> findById(ID id);  
  
    boolean existsById(ID id);  
  
    Iterable<T> findAll();  
  
    long count();  
  
    void delete(T entity);  
  
}
```



Spring Data JPA - Query

Query methods are methods that find information from the database and are declared on the repository interface.

```
public interface PersonRepository extends Repository<Person, Integer> {  
  
    List<Person> findByLastname(String lastname);  
    List<Person> findByEmailAddressAndLastname(String emailAddress, String lastname);  
    // Enables the distinct flag for the query  
    List<Person> findDistinctByLastnameOrFirstname(String lastname, String firstname);  
    // Enabling ignoring case for an individual property  
    List<Person> findByLastnameIgnoreCase(String lastname);  
    // Enabling ignoring case for all suitable properties  
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
    // Enabling static ORDER BY for a query  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  
}
```



Spring Data JPA - CrudRepository

```
@Repository
public interface TicketRepository extends CrudRepository<Ticket, Integer> {
    ...
}
```



Spring Data JPA

You can also use `@Query` and `@NamedQuery`

```
@NamedQuery(name = "User.findByUserName",  
    query = "select u from User u where u.userName = ?1")  
@Table  
@Entity  
public class User {...}
```

```
public interface UserRepository extends JpaRepository<User, Integer> {  
  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
  
    User findByUserName(String userName);  
}
```

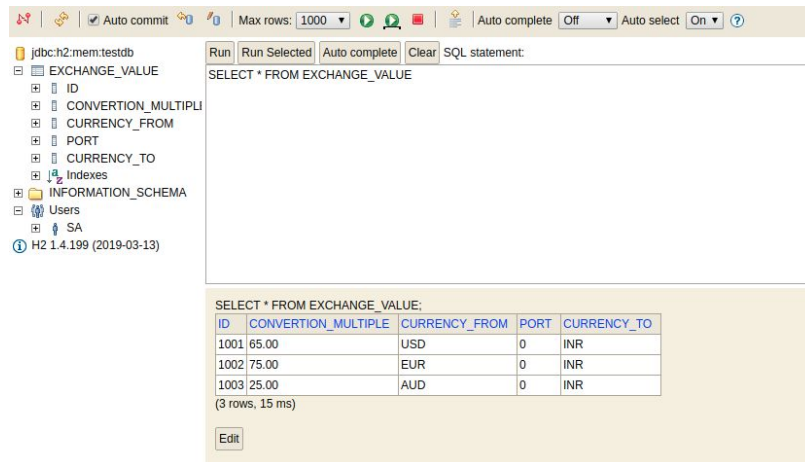
H2 Database: Example

```
@Entity
//http://localhost:8000/h2-console
//verify that jdbc:h2:mem:testdb
public class ExchangeValue {

    @Id
    private long id;

    @Column(name = "currency_from")
    private String from;

    @Column(name = "currency_to")
    private String to;
    private BigDecimal conversionMultiple;
    private int port;
}
```



The screenshot shows the H2 Database console interface. On the left, a tree view displays the database structure, including the 'EXCHANGE_VALUE' table with columns: ID, CONVERSION_MULTIPLE, CURRENCY_FROM, PORT, and CURRENCY_TO. The right pane shows the SQL statement 'SELECT * FROM EXCHANGE_VALUE' and its results. The results table has 3 rows and 5 columns: ID, CONVERSION_MULTIPLE, CURRENCY_FROM, PORT, and CURRENCY_TO. The data rows are: (1001, 65.00, USD, 0, INR), (1002, 75.00, EUR, 0, INR), and (1003, 25.00, AUD, 0, INR). The status bar indicates '(3 rows, 15 ms)'.

ID	CONVERSION_MULTIPLE	CURRENCY_FROM	PORT	CURRENCY_TO
1001	65.00	USD	0	INR
1002	75.00	EUR	0	INR
1003	25.00	AUD	0	INR

```
insert into exchange_value (id,currency_from,currency_to,conversion_multiple,port) values (1001,'USD','INR',65,0)
insert into exchange_value (id,currency_from,currency_to,conversion_multiple,port) values (1002,'EUR','INR',75,0)
insert into exchange_value (id,currency_from,currency_to,conversion_multiple,port) values (1003,'AUD','INR',25,0)
```



Additional Reading

- <https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa>
- <https://docs.spring.io/spring-data/rest/docs/current/reference/html/>
- <https://spring.io/projects/spring-data>
- <https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction-to-query-methods/>



Challenge

You have come this far... Now it's time to apply everything we learned!

For this project, you will have to:

- Create **1 REST APIs** with **at least** 2 entities (for example Students and Courses).
- Create **CRUD endpoints** using the **HTTP standards**
- Create one database that store the info of the related entities
- Using the endpoints created access the information in the database using Spring DATA

Commit the whole project in github

A person's hand is holding a smartphone over a desk. On the desk, there is a laptop, a tablet displaying a bar chart, and a cup of iced coffee with a straw. The entire scene is overlaid with a semi-transparent teal filter.

We are ready!

» **Globant**