OOP in PL/SQL, Oracle Advanced Queues and Java applications in Tomcat container

Sabin TARBĂ
Faculty of Cybernetics, Statistics and Economic Informatics
Bucharest University of Economic Studies
sabin.tarba@csie.ase.ro / sabintarba01@gmail.com

Today, modern and quality software development involves the integration of multiple IT solutions or products, as well as efficient programming paradigms that offer many development opportunities. This scientific article will be based on the separate description and integration of three powerful software products and components: Object-Oriented Programming in Oracle PL/SQL, advanced message queues in Oracle Databases, and Java applications with connectivity to Oracle databases running in an Apache Tomcat container. The PL/SQL procedural language has evolved greatly in recent years, allowing Oracle databases to keep pace even with the evolution of artificial intelligence and machine learning techniques. It includes object-oriented programming features and programming methodologies, so that programmers can define and work with abstract data types that implement the basic notions of object-oriented programming: encapsulation, inheritance, and polymorphism. Oracle Advanced Queues provide an advanced and powerful asynchronous message processing mechanism, high performance and efficient ways to manage data flows. Java applications and the Tomcat container represent a novel choice especially for web application development, but not only, being widely used in server-to-server communications between various systems.

Keywords: *Object-oriented programming, message queues, Java, application, table, type, Oracle, PL/SQL, SQL.*

1 Introduction

Today, modern software development involves the integration of multiple IT solutions or products, as well as efficient programming paradigms that offer many development opportunities. This scientific article will be based on the separate description and integration of three powerful software products and components: Object Oriented Programming in Oracle PL/SQL, advanced message queues in Oracle Databases, and Java applications with connectivity to Oracle databases running in an Apache Tomcat container. The integration of the three technologies helps in the development of scalable, secure, and high-performance enterprise applications [1]. It ensures continuous processing and control over data flows and improves response times to various components in a complex application. Given that a such an application that is built using the integration of the three aforementioned technologies is made in an Oracle ecosystem, more precisely the transactional use of Oracle databases, offers increased security and extensive programming opportunities that it makes available: external tables, indexes for performance, working with files, cryptography, API exposure, complex mechanisms for queries and useful execution plans and so on. Oracle databases have been among the most performing for a long time and are still at the top. Currently, they occupy 1st place according to DB Engines [11], and the most common type is the relational one, but with the multi-dimensional model also available. In the era of accelerated digitalization, software applications are no longer isolated entities, but components of complex, distributed and

interconnected IT ecosystems. Modern software development involves not only creating functional code, but also ensuring a scalable infrastructure, a well-defined architecture and efficient integration between various technologies, programming languages and platforms.

Organizations are facing major challenges such as securing large volumes of data, information security, optimizing performance and guaranteeing operational continuity. In this context, choosing the right technologies and integration models becomes essential for the success of any large-scale software project.

A major trend in enterprise application development is the orientation towards modular, service-based architectures (such as SOA or microservices), which allow for a clear separation of responsibilities, component reuse and easy maintenance [2]. At the same time, management languages and database management systems must collaborate closely to provide high performance and a fluid user experience.

Especially in Oracle ecosystems, the emphasis is on the unified use of advanced features made available by the platform: from PL/SQL as a procedural language with object-oriented support, to internal asynchronous messaging systems and efficient connectivity with distributed Java applications. These components are not just independent technologies, but essential pieces in a coherent mechanism, capable of ensuring robust, secure and optimized processes. At the same time, the adoption of modern programming paradigms – such as object-oriented programming or reactive processing – plays an important role in improving the application structure, code clarity and extensibility. It is not enough for an application to work; it must work efficiently, be easy to adapt and withstand technological and business changes over time.

2. Object-oriented programming in PL/SQL

Object-oriented programming is the most widespread programming paradigm and is the basis of many complex computer systems around the world. It is based on the concept of objects, each of which contains data which are called *fields*, *attributes* or *properties*. They also present methods that represent the way of exposing the functionalities of the object (class) to the calling programs. Object-oriented programming in Oracle PL/SQL is built on the basis of user-defined types. The respective types, also called object types, can have functions that are equivalent in OOP terms to methods that return a result or procedures that have the equivalent of methods that do not return anything and rather perform an action (void) [3], [4]. The example below represents the declaration of a simple "class" with a method (procedure). The declaration is done in two stages and involves the declaration of specifications (spec) and the body (body):

```
create or replace type employee as object (
  id number(19),
  name varchar2(100),
  salary number,

  member procedure print_details
);
/

create or replace type body employee as
  member procedure print_details is
  begin
  dbms_output.put_line('Employee : '||
name || ', salary: ' || salary);
  end;
end;
/
```

Object types also allow the addition of constructors with various parameters as needed. *Constructors* are defined as functions whose name is identical to the object name, but are also declared as a constructor and

return self, the equivalent of this in modern OOP-based languages:

```
create or replace type employee as object (
  id number(19),
  name varchar2(100),
  salary number,

  constructor function employee (self in out nocopy employee, name varchar2, salary number) return self as result,
  member procedure print_details
);
/
```

Instantiation of objects can be done by calling the constructor directly or using the *new* keyword together with it*:*

```
declare
  l_employee employee;
begin
  l_employee:=new employee('Sabin Tarba', 11000);
  l_employee.print_details;
end;
/
```

Static methods are also supported by mentioning static when declaring the function or procedure. Inheritance is partial and is not as flexible as in a language built for use only with objects, such as Java. Inheritance in PL/SQL is based on declaring types under other types using *under*:

```
create or replace type developer under employee (
  overriding member procedure print_details;
);
/
```

There are no interfaces or abstract classes, and no garbage collection, but objects are automatically cleaned up when the PL/SQL block ends or the calling program ends. These limitations can influence the design of complex applications and require a careful approach to resource management. The lack of interfaces means that behavioral contracts cannot be defined without actually implementing inheritance, and the absence of abstract classes can lead to the creation of implicit implementations that may be desirable to remain abstract.

The use of object-oriented programming in PL/SQL, although with certain peculiarities and limitations compared to purely object-oriented languages, offers significant benefits in organizing and code maintainability. It allows developers to model real-world entities more intuitively, reduce code duplication through inheritance, and manage the complexity of large systems through encapsulation and polymorphism. This helps create more robust databases that are easier to extend and maintain over the long term.

Within complex applications developed in Oracle PL/SQL, there is often a need to model and process heterogeneous data, coming from various sources and requiring specific logic for each category of requests. This functional diversity often leads to code fragmentation, which becomes difficult to maintain, extend, or adapt to the dynamic needs of the system. To effectively manage this complexity, an advanced technique can be used, inspired by the principles of metaprogramming and polymorphism in object-oriented languages, namely the implementation of a *type factory* architecture that involves creating objects of certain types based on a common interface [5], but which can also have specific attributes and methods at the child type level. The implementation of such architecture can start from the creation of a special table that can be used to map API identifiers (*operations*) and PL/SQL type objects. An example of such a table can be the representation below:

```
create table type_factory (
  operation varchar2(30),
  service_type varchar2(30)
);
```

Both columns are *30 bytes* in size so that some general rule of naming operations (APIs) and their types is maintained. This

helps maintain readable, clear code and allows compliance with a standard among all developers who will work on the respective application or applications. The *operation* column has the role of specifying an identifier of an operation in the system such as the name of an API or an action taken by the user. A pertinent example could be *auth_login* which would represent the action of a user's authentication in the system. The *service_type* column represents the logical and conceptual mapping of the operation with a PL/SQL type defined by the developer that implements the logic required for that operation (name and user verification, verification if the user is active and others). An example of such a PL/SQL type could be *SRV_AUTH_LOGIN*.

The high degree of similarity in the names of the two components can be observed, as it would be recommended to have some rules for naming these components, not just their size restriction. In the current example, the rule could be to prefix the operation name with *SRV_* which could be the abbreviation for service. Another representative example could be *API_* which would suggest that the respective type serves as an interface between the user, respectively the frontend application, and the logic and the database. In addition to the two columns, other identifiers or other columns can be added to help implement a dynamic mechanism. The main advantage of such an architecture is that it allows the implementation of a logic and an entry point into the database which can be represented by opening a *JDBC* connection from a Java application that calls a stored procedure or a procedure from a package and allows the isolation of the common code that is based and focused on the object interface from the rest of the implementations of the child types. This is very easy and allows for great scalability of systems so that developers can effortlessly maintain the code, there are no dependencies, and the functionality of the application can be very easily extended.

Such an entry point into the database can be represented by a PL/SQL package containing a special procedure for implementing the interface logic and can be considered the example below:

```
create or replace package api_main_pkg is
  procedure on_request(p_request in varchar2, p_response out varchar2);
end;
/

create or replace package body
api_main_pkg is
procedure on_request(p_request in varchar2,
p_response out varchar2) is
  l_service_t service_if;
  l_service_type varchar2(30);
  l_operation varchar2(30);
begin
/*
        Logs
        Another necessary actions or calls
        Parse p_request as a json object and
retrieve l_operation
*/
select t.service_type into l_service_type
from   type_factory t
where  t.operation = l_opeartion;

execute immediate 'begin :x := new ' ||
l_service_type || '; end;' using in out
l_service_t;

l_service_t.example_procedure1();
l_service_t.example_procedure2();
l_serivce_t.example_procedure3();
/*
        Next, the logic can be executed
        Catching exceptions
        Finally, retrieving the response in the
out parameter p_response
*/
p_response := l_service_t.response_json();
end;
end;
```

/

The *on_request* procedure in the *api_main_pkg* package can be called through a *JDBC* connection from a *Java* application that can be deployed in a *Tomcat* container and that can serve as an API between the application that consumes this API, usually the frontend application, and the database. Such an implementation is exemplified in the following chapters. The procedure has an input parameter specified by the in keyword called *p_request* of type *varchar2* which has the maximum size in PL/SQL by default *32767 bytes*. Depending on the need and the large volume of data that could appear in the request, the developer can opt to use the *clob* data type. The representation of the content of this variable could be a *JSON object*, for example:

```
{
   "operation": "auth_login",
   "payload": {
        "userName": "sabintarba",
       "password": "DDSJA421214"
     }
}
```

The operation field represents the API operation and is used to dynamically instantiate the child PL/SQL object through the concept of dynamic SQL. The payload field represents the actual request body that will be used to implement the logic and retrieve the necessary attributes, in this case the user authentication in the system. Retrieving the PL/SQL object type is done through the select into command, and the *l_service_type* variable is of interface type. The creation of the child object is done through dynamic SQL, more precisely through the *execute immediate using in out l_service_type* sequence. After executing this line of code, *l_service_type* remains of interface type, but at the time of code execution (*run time*) it will store a parent type object, in the case described above *SRV_AUTH_LOGIN*. Next, the example procedures and other procedures or logic will be run that will represent either procedures/functions declared and implemented at the interface level, or procedures/functions declared at the interface level with a certain implicit implementation or not and overriding at the child type level.
After executing the logic and all logical steps, at the end of the *on_request* procedure call, a JSON object will be stored in the *p_response* variable to be sent to the application consuming the respective API. An example of a successful response could be:

```
{
   "status": "success", "payload": {
   "sid": "SIDAD14151DASDFG511"
     }
}
```

An example of an error response could be:

```
{
   "status": "fail",
   "payload": {
      "errorMessage": "Invalid credential."
   }
}
```

3. Advanced message queuing mechanisms in Oracle

In the era of distributed applications and enterprise systems, the asynchronous communication component is vital. Oracle Advanced Queuing (AQ) is an integrated solution in Oracle Database that allows the implementation of a queue-type mechanism for managing message flows, asynchronous processing and decoupling application logic [6]. This mechanism is based on principles inspired by messaging architecture and is implemented through native PL/SQL facilities. Queues in Oracle are part of Oracle AQ – a messaging system integrated in Oracle Database that allows sending and receiving messages (objects, data) between applications, sessions or components. They work according to the *FIFO (First In, First Out)* principle. There are actions for enqueuing,

also called *enqueue*, and for removing from the queue, called *dequeue*. The creation of queues and their management is done with special packages that need execute rights, enqueue and dequeue privileges and the *aq_administrator_role* role for the current Oracle user so that he can use them. A big advantage of them is that they can be used with persistent saving in the database or they can be volatile, and the data saved in RAM. There are queue tables and message queues. The relationship between these is 1:N, a queue table can have several queues working on it. Creating a queue table is done by calling dbms_aqadm.create_queue_table(...). Queue creation instead is accomplished by dbms_aqadm.create_queue(queue_table => <previously created table>…). Immediately after creation, the queue is not started automatically, but the start call will have to be made, namely dbms_aqadm.start_queue(...). Sending a message to the queue, the *enqueue* operation, is accomplished by dbms_aq.enqueue(...), and removing it from the queue, the dequeue operation, is performed by dbms_aq.dequeue(...). Advanced Queues in PL/SQL are an extremely powerful and scalable solution for applications that require asynchronous communication, message flow management, or component decoupling [2], [7]. Although sometimes overlooked, AQ offers a viable alternative to external messaging systems (RabbitMQ, Kafka), with the advantage of native integration into Oracle Database and support for full transactionality.

Oracle Advanced Queuing provides a highly flexible asynchronous communication infrastructure that is perfectly suited to enterprise applications that require coordination between distributed components.

One of the key aspects of this mechanism is its support for transactionality. Enqueued messages can participate in regular SQL transactions, which means that if a transaction fails, the message enqueued in that context will not be preserved. This guarantees data consistency and application integrity. For example, in a scenario where a message is sent to a queue as part of a data update, only if all operations in the transaction succeed will the message be effectively placed in the queue and available for further processing. This behavior is crucial in mission-critical applications where message loss or incorrect processing can have significant effects.

In addition, AQ offers sophisticated queue configuration options. You can define message prioritization rules, create multi-consumer queues, where multiple sessions can retrieve messages from the same queue based on filters or subscription rules. There is also support for notification, so that when a message arrives in the queue, a handler can be automatically notified to start processing, eliminating the need for constant polling. These notifications can be transmitted via PL/SQL callbacks or even through integration with Oracle Event Service, depending on the chosen configuration.

Another valuable aspect is the support for complex payloads. Messages can contain user-defined Oracle objects, JSON structures, or simple SQL types. Thus, AQ is not limited to sending strings or primitive values, but can transport complex data structures, tailored exactly to the requirements of the application. There are also facilities for automatic message scheduling – a message can be scheduled to be available for dequeue only after a certain period of time, thus providing support for delayed or scheduled tasks.

In production environments, AQ can be used in tandem with other Oracle technologies, such as Oracle Streams or Oracle Data Guard, for message replication and ensuring robust communication in distributed architectures. AQ can be secured through granular access policies, so that only certain roles or users can enqueue or dequeue on certain queues, using privileges and auditing poli-

cies. AQ integration with Oracle Enterprise Manager allows for queue monitoring, identifying bottlenecks or delays in message processing, and providing prompt alerts in case of performance or availability issues.

Another important advantage is the persistence of messages. Even in the event of a system crash or uncontrolled database shutdown, persistent messages in the queues will be preserved and can be resumed when the database is restarted. For critical applications that cannot afford to lose messages or duplicate processing, this is vital. There is also the possibility of defining message expiration policies – a message that is not processed within a certain interval can be automatically removed from the queue or sent to an exception queue for further processing or manual analysis.

In practice, AQ is successfully used in banking applications, ERP systems, logistics applications or any other complex system that involves many independent modules that need to collaborate in an efficient and scalable way. Queues thus allow for a loosely coupled architecture, in which components do not need to know internal details about each other and can be developed, tested and scaled independently. AQ proves to be an extremely mature solution, being available and supported in Oracle Database for more than two decades, and continues to evolve, now also offering integration options with Oracle Cloud Infrastructure, Oracle Streams AQ or even hybrid integration services. Another essential aspect of Oracle Advanced Queuing is the support for parallel message processing and integration with Oracle job scheduling mechanisms. Thus, applications can benefit from a distributed processing, where multiple concurrent sessions or processes can simultaneously retrieve and process messages from the same queue. This behavior is controlled and optimized by setting queue attributes, such as multiple consumers, retention, *delivery_mode*, and *dequeue_mode*. The way messages are distributed to consumers can also be customized to avoid collisions or double processing. Oracle provides a balance between performance and consistency through these mechanisms, even in complex scenarios with high message traffic and intensive processing.

Furthermore, AQ also allows the definition of subscriber rules, sophisticated rules based on PL/SQL expressions or SQL conditions that determine whether a particular message should be delivered to a particular consumer. This filtering-based model allows for advanced message routing scenarios, without the need to duplicate logic in the client application. In other words, applications can leave the routing and filtering logic to AQ, focusing only on the actual data processing. For example, in a multi-tenant system, each message can contain a tenant identifier, and the queue can be configured to deliver messages only to consumers associated with that tenant.

AQ also offers compression and space optimization options, important in scenarios where large volumes of messages are managed or applications must comply with strict resource usage policies. It can also be integrated with *Oracle RAC* (Real Application Clusters) infrastructure, allowing horizontal scaling and high availability of queue services. In a *RAC* environment, multiple Oracle instances can participate in concurrent enqueue and dequeue without compromising message integrity or delivery order, which is a critical feature for applications with strict uptime requirements.

Not to be overlooked is AQ's ability to work in tandem with external systems. Through the gateways and connectors available in Oracle SOA Suite or through Oracle's Java and JMS (Java Message Service) APIs, messages in queues can be sent or received from Java applications, enterprise middleware, or even web services. This interoperability capability paves the way for AQ integration into hybrid architectures, where in-

ternal Oracle queues can work with external solutions such as IBM MQ, ActiveMQ, RabbitMQ, or even modern cloud-based services such as Azure Service Bus or AWS SQS. This flexibility greatly extends the utility of AQ, making it a viable solution not only for the Oracle ecosystem, but also for the broader, modern, and diverse IT landscape.

4. Integrating PL/SQL and Oracle Queue OOP mechanisms with Java applications using Apache Tomcat

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It has evolved from humble beginnings to power much of today's digital world, providing the trusted platform on which many services and applications are built. New, innovative products and digital services designed for the future continue to be based on Java. In the context of modern software development, interoperability between components implemented in different programming languages is becoming a fundamental element. This paper proposes an integrated approach in which business logic, implemented in PL/SQL with support for object-oriented programming, is combined with the Oracle Advanced Queuing (AQ) queue mechanism and is connected to Java web applications deployed on the Apache Tomcat server. Connecting a Java application to an Oracle database can be done through a driver. The best known is *ojdbc8.jar* (Java DataBase Connectivity) which includes APIs and facilitates connection to a database [8]:

```
import java.sql.*;
public void executeProcedure() {
Connection connection =
DriverManager.getConnection("jdbc:oracl
e:thin:@<ip>:<port>/<sid>", "<username>",
"<password>");
CallableStatement statement = connec-
tion.prepareCall("<call procedure>");
statement.execute();
statement.close();
connection.close();
}
```

The above call represents opening a connection to a database, executing a procedure and very importantly closing the resources by calling the close method. Configuring the Java application connection to an Oracle queue is more complex. First of all, if you want a continuous process to run and listen for new messages added to the queue, the class that will implement this must be of type Thread and implement the *run()* method inside which the listener for the Oracle queue will have to be registered through support classes such as *AQNotificationRegistration* and the *addListener* method call. On the other hand, the *onAQNotification* method will also have to be implemented which will represent the section of code that will be executed when a new message is put in the queue.

Apache Tomcat is an open source web server and servlet container, developed by the *Apache Software Foundation* (ASF), used to run web applications written in Java [9]. It implements the Java Servlet, JavaServer Pages (JSP) and WebSocket specifications, providing an easy and efficient solution for developing and running applications based on Java Enterprise technology. Unlike more complex application servers such as JBoss, like WebSphere or WebLogic, Apache Tomcat specializes in handling HTTP requests and Java web components, without including advanced features such as Enterprise JavaBeans (EJB). This makes it faster, easier to configure, and suitable for small to medium-sized web applications. Apache Tomcat is an efficient and easy-to-use solution for running Java web applications. With its low resource consumption and simple configuration, it is

ideal for developers who want to quickly deploy web applications without the complexity of a full enterprise application server. Tomcat also integrates easily with databases, messaging technologies (such as Oracle AQ), and modern frameworks such as Spring Boot, making it a versatile tool for developing scalable web applications.

Integrating Java applications running on Apache Tomcat with PL/SQL and Oracle Advanced Queuing (AQ) mechanisms requires not only appropriate Java code, but also careful configuration of the Tomcat server. This configuration has as its main purpose the establishment and management of the connection to the Oracle database in a centralized, reusable and efficient way, avoiding frequent opening and closing of connections directly in the application code. Thus, a data source (*DataSource*) is defined within Tomcat, which is then used by the application components to interact with the database and implicitly with the AQ queues [1], [10]. The data source configuration in Tomcat is done in the context.xml file, located in the conf directory or inside the application in the META-INF/context.xml file. In this file, a Resource of type *javax.sql.DataSource* is declared, which specifies the connection parameters, the connection pool and other attributes relevant for performance and stability, such as the maximum pool size, the waiting time for obtaining a connection or the connection validation policy.

After configuring the data source, the Java application deployed on Tomcat can obtain connections via JNDI (*Java Naming and Directory Interface*), without having to manually manage login details in code.

This increases security and allows changing the database configuration without recompiling the application. In scenarios involving active listening to an AQ queue, the component running in the background within the web application can use this data source to open a persistent connection to the Oracle database, which is necessary to register a listener for the desired queue. In addition to defining the data source, it is also recommended to configure appropriate logging at the Tomcat level to monitor the interaction with the database and any connection issues, deadlocks, or timeouts. This can be done by configuring Tomcat's logging files (e.g. *logging.properties*) and by integrating with logging frameworks used by the application.

Another important aspect in the context of applications that listen to events or process messages from Oracle queues is the correct management of the lifecycle of components that connect to the database. It is important that the initialization of connections to queues, the registration of listeners and the starting of listening threads be done in special methods, such as *init()* within a *ServletContextListener* or in the *@PostConstruct* methods of *Spring* components, and their termination be handled in *destroy()* or *@PreDestroy* methods, to ensure the correct release of resources when the application is closed or the server is restarted.

In production environments, where several applications run simultaneously on the same Tomcat server and access the Oracle database, it is essential to correctly isolate data sources, clearly define JNDI contexts for each application and apply strict security policies, including restricting access to the application's JNDI context.

Correct *Tomcat* configuration plays a central role in guaranteeing a stable and high-performance communication channel between Java applications and Oracle components that implement business logic and advanced queue mechanisms. In this way, a robust, modular and scalable architecture is created, capable of supporting complex enterprise applications, with strict requirements for reliability, performance and asynchronous communication. The Tomcat container can be configured in single instance mode,

which means that there will be *CATALINA_HOME* (system variable that holds the installation path) and *CATALINA_BASE* (system variable that holds the working path) will have the same value. In a *multi-instance* architecture, there will be a single *CATALINA_HOME* and several *CATALINA_BASE* depending on the number of instances. Each instance directory will contain the necessary Tomcat configuration files (*web.xml, context.xml etc*) and the necessary directories such as webapps. Each instance can be associated with a *systemctl* service when started, depending on the instance that is wanted to be started, the corresponding *CATALINA_BASE* will be set.
Example of configuration:
CATALINA_HOME = /opt/tomcat CATALINA_BASE = /srv/tomcat/inst1 CATALINA_BASE = /srv/tomcat/inst2

Extending the Tomcat configuration to fully support Oracle AQ integration also involves considering advanced aspects related to resource management, security, performance optimization, and scalability. A well-configured Tomcat server provides not only a stable platform for running the web application, but also a centralized control point for all connections to the Oracle database, which is essential in the context of a distributed system that uses asynchronous messaging mechanisms.
An important first step in this direction is the configuration of the connection pool (*Connection Pooling*), which must be appropriately sized depending on the expected message volume, the number of concurrent users and the type of operations performed. Attributes such as *maxTotal*, *maxIdle*, *minIdle*, *maxWaitMillis* and *validationQuery* should be carefully tuned to avoid resource exhaustion situations, but also to prevent message processing deadlocks. Validating idle connections via *testWhileIdle* or *testOnBorrow*, along with configuring a simple validation query such as *SELECT 1 FROM DUAL*, helps maintain pool stability.
In terms of application availability and reliability, it is recommended that the AQ listener implemented in the Java application be registered and initialized at a controlled point in the application lifecycle. In Tomcat, this can be achieved using a *ServletContextListener*, which allows background components (such as threads listening to Oracle queues) to be started at application initialization and stopped in a controlled manner at application shutdown. This avoids losing the connection to the queue or situations where threads remain active after the application shutdown, which could lead to memory leaks or deadlocks.
For applications running in enterprise environments with multiple Tomcat instances (for example, in load balancing or failover scenarios), queue access synchronization is required to prevent duplicate message processing. Oracle AQ allows the configuration of the dequeue mode with various policies (for example, REMOVE, BROWSE, LOCKED) and allows the specification of a concurrency strategy for message delivery. The application must ensure that there are no multiple consumers simultaneously for the same queue, unless this behavior is intentional and the queue is properly configured for concurrent distribution (multi-consumer queue).
On the security side, Tomcat must be configured to securely manage the credentials used to connect to the database. Instead of directly defining the user and password in context.xml, mechanisms such as external JNDI Environment Entries, integrations with secure password managers or even encrypting configuration files and decrypting them dynamically at server startup. These practices are essential for protecting sensitive data, especially in applications operating in fields such as finance, government or healthcare.

Another important aspect is monitoring. Tomcat offers various options for monitoring the performance of data sources, the number of active connections, response times and possible errors. By integrating with external monitoring solutions (such as Prometheus, Grafana, Datadog or ELK Stack), administrators can have a clear picture of the application's behavior in real time and can detect potential connectivity issues, queue saturation or message blocking early.

If the application is based on modern frameworks such as Spring or Jakarta EE, Tomcat configurations can also be dynamically completed or managed through the configuration files of these frameworks, allowing the injection of data sources defined in Tomcat directly into the application components. Thus, integration becomes smoother, code becomes cleaner, and the component lifecycle is better controlled.

In conclusion, a well-configured Tomcat is not limited to simply running a Java application, but becomes a reliable, high-performance and scalable integration platform for interacting with Oracle components and advanced queue-based messaging systems. It actively supports business logic implemented in PL/SQL, facilitates asynchronous communication between components and plays a key role in orchestrating processes in a complex software ecosystem. This modern architecture, which combines the power of the Oracle database with the flexibility of Java and the efficiency of Tomcat, provides a complete solution for enterprise applications that require reliability, modularity and extensive interoperability. Beyond these core capabilities, Tomcat acts as a strategic middleware layer that enables the seamless exposure of backend services through RESTful APIs, user interfaces, and administrative tools. This supports not only internal operational flows but also integration with external systems, mobile applications, and third-party services, making the platform suitable for a wide range of business scenarios. Through standardized protocols and interface abstraction, Tomcat ensures that applications remain adaptable and compatible with evolving technologies, business requirements, and integration standards.

Furthermore, this architectural approach supports the principles of service-oriented and event-driven design, allowing for a clean separation of concerns, improved testability, and easier maintainability. The asynchronous processing made possible through Oracle Advanced Queuing enhances system responsiveness under heavy workloads, while reducing tight coupling between modules. As a result, organizations can scale individual components independently, introduce new features with minimal risk, and maintain high levels of availability and performance, even as system complexity grows.

Ultimately, by harnessing the strengths of each technology—Oracle's robust data and messaging capabilities, Java's portability and object-oriented design, and Tomcat's lightweight yet powerful application serving—this integrated platform empowers developers to build enterprise-grade applications that are not only technically sound, but also strategically aligned with long-term goals of scalability, agility, and resilience in an increasingly connected digital world.

5. Extended conclusions

The coherent integration of all the aforementioned technologies allows the creation of an enterprise-type information system with a high level of complexity, capable of managing large volumes of data, providing high performance and being easy to scale. Such an architecture not only supports the technical requirements of modern applications, but also responds to business needs through flexibility, modularity and operational reliability.

An essential element in this construction is the use of object-oriented programming in PL/SQL. This allows the structuring of business logic in a clear, coherent and reusable way, through object types, member methods and PL/SQL packages. This modular organization favors the maintenance, unit testing and extensibility of the application components, similar to the way modern languages such as Java or C# allow the development of object-oriented components. Within an enterprise system, this ability to logically delimit components and to test or update them independently is essential to maintain the quality and consistency of the entire application in the long term.

Oracle Advanced Queuing (AQ) also brings significant added value by providing a native mechanism for asynchronous processing and decoupled communication between application components. By using queues, processes can exchange information without the need for strict synchronization between sender and receiver, making the system more resilient to errors and performing better under high load conditions. Queues can be used to model complex workflows, process tasks in parallel, or manage communication between heterogeneous subsystems, all of which contribute to the scalability and reliability of the overall system.

At the presentation and web interface layer, using an application server such as Apache Tomcat, along with Java technologies such as Servlet, JSP, or even modern frameworks such as Spring MVC, provide a stable and extensible platform for developing intuitive graphical interfaces and REST APIs. They allow not only end-users access to application functionalities, but also easy integration with other systems through standardized protocols. In addition, integration with the Oracle database via JDBC provides an efficient way to access and manipulate objects defined in PL/SQL, thus allowing direct collaboration between Java logic and logic stored in the database.

This architectural approach, based on the principles of modularity, decoupling, reuse and scalability, aligns perfectly with modern software architecture standards, such as SOA (Service-Oriented Architecture) and microservices. A major advantage of this solution is that it does not require complex external infrastructure (such as Kafka, RabbitMQ, EJBs, etc.), thus reducing implementation and maintenance costs, without compromising on the capabilities offered. Extensive use of native Oracle facilities – such as object types, queues, packages, automated jobs and security policies – ensures superior performance by executing logic directly in the database engine, without the need for costly intermediaries.

Such an integrated and object-oriented software architecture allows building robust, secure and efficient enterprise applications, suitable for a wide range of critical domains: from ERP and CRM systems, to educational platforms, financial applications, reservation systems, government applications or customized solutions for large organizations. This strategy offers an optimal balance between technological innovation, solution maturity and operational efficiency, thus contributing to the development of sustainable and adaptable applications over time. In addition to the technical and architectural benefits, such an integrated solution also significantly contributes to reducing the total cost of ownership (TCO). By extensively using technologies already existing Oracle and Java ecosystem – without introducing expensive or difficult-to-maintain external dependencies – optimizes both development and long-term operation and support efforts. Organizations can thus maximize the value of existing infrastructure, avoid the complexity of unnecessary integrations, and focus on developing functionality specific to their field of activity.

Moreover, this architecture allows easy adaptation to the constantly changing requirements of the business environment. The modular structure of the application, based on objects, packages and queues, allows the extension of functionalities without affecting the existing components. For example, new message types can be added to the AQ system to support new business processes, or new web modules can be created in Tomcat to expose additional REST services. These changes can be made incrementally, with minimal impact on the existing functionalities, which is essential for maintaining operational continuity in an enterprise environment.

Also, the possibility of automating repetitive or critical tasks through Oracle jobs (e.g. periodic data processing, cleaning, aggregation, notifications) contributes to increasing operational efficiency and reducing the risk of human error. Thus, the system not only promptly responds to user requests or other applications, but can also autonomously initiate internal processes based on predefined rules or conditions. This transforms the application into an intelligent ecosystem, capable of operating optimally both reactively and proactively. In a context in which the requirements for data security, traceability and auditability are becoming increasingly strict, the proposed approach also responds to these challenges. Oracle offers advanced access control, logging and auditing mechanisms, which can be applied directly to the PL/SQL objects, packages and procedures used in processing. Thus, the application can guarantee not only the integrity and confidentiality of data, but also the ability to demonstrate at any time who, what, when and how operated in the system – essential aspect in regulated industries such as finance, healthcare or government. In addition, this architecture provides an excellent framework for implementing clear versioning and configuration management policies. Being based on distinct and well-delimited components – such as PL/SQL packages, Java classes and web modules – each element can be managed separately in terms of version, code changes and life cycle. This is extremely important in enterprise projects, where implementations are usually done incrementally, and development, testing and operations teams must collaborate in a coordinated way. Through efficient source code management and by automating the build and deployment processes (e.g. using Maven, Jenkins or other DevOps tools), a high level of consistency and predictability can be achieved, reducing risks and time to production.

The flexibility of this architecture also allows the adoption of modern principles such as continuous integration and continuous delivery (CI/CD), which offers the advantage of fast development cycles, automated testing and frequent delivery of new functionalities to users. By organizing business logic into reusable and testable components in isolation, a stable and robust, yet agile ecosystem can be built, capable of quickly responding to changes required by the business environment. In addition, due to the decoupled nature of the components, teams can work in parallel on different modules without causing major blockages or interference between functionalities. Another major benefit is the architecture's ability to support detailed auditing and reporting of operations. Every step in the business process, from receiving the request in the web application to processing messages in PL/SQL and updating the database, can be monitored and recorded. The flexibility of this architecture also allows the adoption of modern principles such as continuous integration and continuous delivery (CI/CD), which offers the advantage of fast development cycles, automated testing and frequent delivery of new functionalities to users. By organizing business logic into reusable and testable components in isolation, a stable and robust, yet agile ecosystem can be built,

capable of quickly responding to changes required by the business environment. In addition, due to the decoupled nature of the components, teams can work in parallel on different modules without causing major blockages or interference between functionalities. Another major benefit is the architecture's ability to support detailed auditing and reporting of operations. Every step in the business process, from receiving the request in the web application to processing messages in PL/SQL and updating the database, can be monitored and recorded.

Finally, the synergy between object-oriented PL/SQL components, Oracle Advanced Queuing mechanism and modern Java platforms, such as Tomcat with Servlets, JSP or Spring MVC / Spring Boot, outlines a solid, flexible and efficient architecture, ideal for developing enterprise applications. This combination allows not only a clear delimitation between the application layers – presentation, business logic and persistence – but also a fluent and secure communication between them, both synchronous and asynchronous.

By making the most of the native capabilities offered by Oracle and Java, the dependence on complex external technologies is significantly reduced, without compromising performance or scalability. On the contrary, a coherent, controllable and easy-to-maintain infrastructure is obtained, capable of responding to both advanced technical requirements and business pressures regarding delivery times, operational costs and compliance requirements.

This architecture is a perfect fit for mission-critical applications such as ERP systems, CRMs, financial management solutions, educational platforms, reservation systems, or any other type of application that requires data-intensive processing, high resilience, low response times, and rapid scalability. Its modularity and decoupled nature ensure that each component can evolve independently, enabling organizations to implement changes, updates, or enhancements with minimal disruption to the system as a whole. This is particularly valuable in environments where continuous availability and system uptime are essential, and where any downtime could result in significant operational or financial losses. Furthermore, the architecture's ability to leverage both synchronous and asynchronous communication models through mechanisms like Oracle Advanced Queuing (AQ) allows for flexible orchestration of internal processes. This enables background tasks, batch processing, and event-driven workflows to coexist seamlessly with real-time transactional operations. As a result, systems built on this model can efficiently handle a wide spectrum of workload scenarios, from high-volume transactional spikes to long-running background jobs, without compromising performance or user experience.

Security and compliance also benefit from this approach. The use of Oracle's robust native security features—such as fine-grained access control, auditing, encryption, and role-based privileges—ensures that sensitive data is handled appropriately and that all operations are traceable. This level of control is crucial for industries that must adhere to strict regulatory frameworks, such as finance, healthcare, or public administration. Combined with structured application-level controls in the Java tier, the architecture supports a defense-in-depth strategy that safeguards critical assets across all layers. Another key strength of this architecture is its extensibility. Organizations can gradually introduce new services, interfaces, and integration points as their needs evolve—whether through RESTful APIs, SOAP services, microservices, or integration with third-party platforms without requiring a major overhaul of the existing system. This future-proof design makes it easier to adopt emerging technologies, adapt to new business models, or scale

operations geographically, all while maintaining a coherent and stable core system.

In a constantly changing technological landscape, where agility, reliability, and performance are increasingly vital to competitiveness, this approach represents a solid strategic choice. It empowers organizations to deliver sustainable, high-performance, and adaptable IT solutions that align with modern architectural principles—such as service orientation, modularity, and scalability—without sacrificing clarity, maintainability, or operational efficiency. Ultimately, this architecture positions businesses to not only meet their current technological challenges but also to anticipate and respond effectively to future demands.

Moreover, by building on widely adopted and well-documented technologies like Oracle and Java, organizations benefit from a rich ecosystem of tools, community support, and skilled professionals. This significantly lowers the barrier to entry for new team members and facilitates long-term maintenance and scalability of the solution. The availability of mature debugging, monitoring, and performance-tuning utilities allows for proactive system optimization and rapid issue resolution, contributing to overall system robustness. In addition, the alignment with established enterprise standards ensures better interoperability with existing systems and easier integration into broader digital transformation initiatives. This combination of technical maturity, ecosystem support, and future-readiness makes the architecture not only a technically sound choice, but also a wise long-term investment for organizations aiming to maintain a competitive edge in a dynamic digital economy.

References

[1] N. A. N. Sobri, M. A. H. Abas, I. M. Yassin, M. S. A. Megat Ali, N. M. Tahir, A. Zabidi, and Z. I. Rizman, "A Study of Database Connection Pool in Microservice Architecture," JOIV: International Journal on Informatics Visualization, vol. 6, no. 2, pp. 566–571, 2022.

[2] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston, MA: Addison-Wesley Professional, 2003.

[3] S. Feuerstein and B. Pribyl, Oracle PL/SQL Programming, 6th ed. Sebastopol, CA: O'Reilly Media, 2014.

[4] Oracle Corporation, "Oracle Database PL/SQL Language Reference, 19c," Oracle Documentation, Doc ID E96448-10, 2021. https://docs.oracle.com/en/database/oracle/oracle-database/19/lnpls/

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1994.

[6] Oracle Corporation, "Oracle Database Advanced Queuing User's Guide, 19c," Oracle Documentation, Doc ID E96442-09, 2021.
https://docs.oracle.com/en/database/oracle/oracle-database/19/adque/

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec, "The Many Faces of Publish/Subscribe," ACM Computing Surveys, vol. 35, no. 2, pp. 114–131, 2003.

[8] M. Fisher, J. Ellis, and J. Bruce, JDBC API Tutorial and Reference, 3rd ed. Boston, MA: Addison-Wesley Professional, 2003.

[9] J. Brittain and I. F. Darwin, Tomcat: The Definitive Guide, 2nd ed. Sebastopol, CA: O'Reilly Media, 2007.

[10] Apache Software Foundation, "Apache Tomcat 10 Configuration Reference: JNDI Datasource Resources," Apache Documentation, 2024.
https://tomcat.apache.org/tomcat-10.1-doc/jndi-resources-howto.html

[11] DB-Engines
https://db-engines.com/en/ranking

Sabin TARBĂ graduated from the Faculty of Cybernetics, Statistics and Economic Informatics of the Academy of Economic Studies of Bucharest, class of 2023, and completed his Master's degree in Databases at the same faculty in 2025. He currently works as a Backend Developer on a fintech/banking platform, building systems on Oracle PL/SQL and Java. His work focuses on system integration, database performance, and building reliable backend services for financial applications and not only.