La arquitectura RDBMS-only

Una arquitectura database-centric para aplicaciones Web

Alfonso Vicente

Programa de Posgrado en Informática
Facultad de Ingeniería
Universidad de la República

Montevideo – Uruguay
Octubre de 2021

La arquitectura RDBMS-only

Una arquitectura database-centric para aplicaciones Web

Alfonso Vicente

Tesis de Maestría presentada al Programa de Posgrado en Informática, Facultad de Ingeniería de la Universidad de la República, como parte de los requisitos necesarios para la obtención del título de Magíster en Informática.

Directores:
Dr. Ing. Prof. Lorena Etcheverry
Dr. Ing. Prof. Ariel Sabiguero

Director académico:
Dr. Ing. Prof. Héctor Cancela

Montevideo – Uruguay
Octubre de 2021

Vicente, Alfonso

La arquitectura RDBMS-only / Alfonso Vicente. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2021.

XIV, 149 p. 29, 7cm.

Directores:

Lorena Etcheverry

Ariel Sabiguero

Director académico:

Héctor Cancela

Tesis de Maestría – Universidad de la República, Programa en Informática, 2021.

Referencias bibliográficas: p. 139 – 149.

1. RDBMS, 2. Arquitectura de Aplicaciones Web, 3. Arquitecturas centradas en la Base de Datos. I. Etcheverry, Lorena, Sabiguero, Ariel, . II. Universidad de la República, Programa de Posgrado en Informática. III. Título.

MEMBERS OF THE THESIS DEFENSE COURT

Dr. Ing. Prof. Juan Carlos Corrales

Dr. Ing. Prof. Laura González

Dr. Ing. Prof. Daniel Calegari

Montevideo – Uruguay
Octubre de 2021

A mi hijo Agustín.

Agradecimientos

Este trabajo no se hubiera realizado sin los buenos oficios de Ariel, que se ocupó de encontrar alternativas a un abandono seguro, debido a una mala decisión del tema en un intento anterior de realizar una tesis de maestría. No es posible insistir demasiado en la ventaja de trabajar en un tema que apasiona. Lorena y Héctor creyeron en la posibilidad de desarrollar la idea central de esta tesis, pese a su rareza. Durante todo el transcurso del trabajo, la orientación, dedicación y ánimo de Ariel y Lorena fueron decisivos. El seguimiento, ánimo y oportunos aportes de Héctor también significaron mucho. Con los tres estoy muy agradecido.

Estoy infinitamente agradecido con mi mujer y mi hijo, por la paciencia frente a las incontables horas que no les dediqué por estar trabajando.

RESUMEN

Las arquitecturas de múltiples niveles han sido el estándar *de facto* para aplicaciones web, dejando poco lugar para arquitecturas alternativas. En la industria existe un producto para desarrollar y ejecutar aplicaciones web que sigue una arquitectura diferente, centrada en el RDBMS al extremo de no necesitar ningún otro componente para funcionar. En la academia no hay muchos trabajos que aborden las arquitecturas centradas en un RDBMS en general, y esta arquitectura extrema en particular no ha sido considerada. En este trabajo se analiza el estado del arte de las arquitecturas centradas en un RDBMS, y se analiza un ejemplo de arquitectura extrema centrada en el RDBMS. Se describe el caso general de la arquitectura que he llamado *RDBMS-only*, y se siguen los lineamientos de esta arquitectura en el desarrollo de un prototipo funcional. En base a la implementación de este prototipo se muestra la factibilidad de la arquitectura para una clase de aplicaciones y se realiza un análisis crítico de la arquitectura.

Palabras claves:
RDBMS, Arquitectura de Aplicaciones Web, Arquitecturas centradas en la Base de Datos.

ABSTRACT

Multi-tier architectures have been the *de facto* standard for web applications, leaving little room for alternative architectures. In the industry there is a product to develop and run web applications that follows a different architecture, centered on the RDBMS to the extreme of not needing any other component to function. There are not many papers in academia that addresses RDBMS-centric architectures in general, and this extreme architecture in particular has not been considered. In this work, the state of the art of database-centric architectures is analyzed, and an example of extreme database-centric architecture is analyzed. The general case of the architecture that I have called *RDBMS-only* is described, and the guidelines of this architecture are followed in the development of a functional prototype. Based on the implementation of that prototype, the feasibility of the architecture for a class of applications is shown, and a critical analysis of the architecture is carried out.

Keywords:
RDBMS, Web Application Architecture, Database-centric Architectures.

Lista de siglas

En este apartado se presenta una lista de las principales siglas utilizadas en esta tesis.

4GL *Fourth-Generation programming Language* 24

ACID *Atomicity, Consistency, Isolation, Durability* 3, 12, 13, 15

ADT *Abstract Data type* 72

AJAX *Asynchronous JavaScript and XML* 25, 28, 35, 58, 62, 64, 88, 110

APEX *Oracle Application Express* 4, 8, 9, 17, 21, 22, 23, 24, 25, 26, 27, 30, 31, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 55, 56, 63, 64, 65, 67, 69, 70, 77, 81, 118, 133, 134, 135, 136

API *Application Programming Interface* 4, 31, 69

BL *Business Logic* 31, 32, 72, 73, 74, 96, 101, 102, 107, 108, 111, 113, 125

C4ISR *Command, Control, Communications, Computers, Intelligence, Surveillance and Reconnaissance* 10

CBMG *Customer Behavior Model Graph* 97

CGI *Common Gateway Interface* 13, 16, 135

CMS *Content Management System* 8

CRM *Customer Relationship Management* 12

CSS *Cascading Style Sheets* 25, 64, 110

DAD *Database Access Descriptor* 45, 47, 48, 65

DBMS *Database Management System* 2, 3, 7, 12, 15, 27

DBPL *Database Programming Language* 3, 5, 17, 21, 33, 72, 76, 77, 124, 125, 136, 137

DL *Data Logic* 31, 32, 72, 73, 96, 104, 105, 106, 107, 122, 125, 137

DML *Data Manipulation Language* 40, 61, 73

DOC *Distributed Object Computing* 16

DR *Disaster Recovery* 44

EAP *Enterprise Architecture Planning* 10

EAST *Evolving Application Synopsis Tool* 23

EJB *Enterprise JavaBeans* 31, 118

EPG *Embedded PL/SQL Gateway* 44

ERP *Enterprise Resource Planning* 12

FEAF *Federal Enterprise Architecture Framework* 11

FOSS *Free and Open Source Software* 77

GUI *Graphical User Interface* 18

HA *High Availability* 44

HTML *HyperText Markup Language* 19, 25, 26, 48, 49, 50, 51, 63, 64, 65, 66, 67, 68, 70, 73, 78, 80, 84, 90, 98, 99, 100, 101, 102, 103, 109, 110, 111, 112, 113, 114

HTTP *Hypertext Transfer Protocol* 4, 11, 12, 15, 18, 20, 22, 34, 40, 41, 44, 47, 48, 50, 58, 63, 64, 65, 66, 67, 68, 69, 76, 77, 78, 79, 80, 81, 82, 83, 84, 86, 88, 89, 90, 101, 129, 130, 131, 135

IAF *Interactive Application Facility* 17

IDE *Integrated Development Environment* 21, 22, 25, 39, 52, 54, 57, 63, 64, 70, 71, 73, 74, 76, 91, 107, 110, 124, 126, 135, 136, 137, 138

JCA *Java EE Connector Architecture* 118

JDBC *Java Database Connectivity* 4, 124, 129

JMS *Java Message Service* 118

JSON *JavaScript Object Notation* 28, 73, 83, 86, 91, 109, 110

JVM *Java Virtual Machine* 20

LCAP *Low-Code Application Platform* 2, 21

LCDP *Low-Code Development Platform* 2

MER *Modelo Entidad-Relación* 11, 40, 85, 92, 94, 100

ML *Machine Learning* 3

MVC *Model–View–Controller* 23, 24, 30, 31

MWOD *Middleware On Demand* 25

NCA *Network Computing Architecture* 18

OCI *Oracle Call Interface* 17

OIDC *OpenID Connect* 118

OMA *Object Management Architecture* 16

OMG *Object Management Group* 16, 38

OODBMS *Object-Oriented Database Management System* 14

OOP *Object-Oriented Programming* 13

ORDBMS *Object-Relational Database Management System* 125

ORDS *Oracle REST Data Services* 44, 118

ORM *Object-Relational Mapping* 15, 30

OWA *Oracle Web Access* 24

PaaS *Platform as a Service* 25

RAD *Rapid Application Development* 17, 21, 23, 26

RAE *Real Academia Española* 12

RDBMS *Relational Database Management System* 1, 3, 4, 5, 8, 9, 12, 13, 14, 15, 17, 20, 21, 22, 23, 24, 25, 27, 29, 30, 32, 33, 34, 35, 36, 37, 38, 40, 44, 45, 51, 63, 65, 71, 72, 73, 76, 77, 78, 81, 82, 85, 88, 89, 90, 91, 93, 95, 103, 106, 117, 118, 119, 120, 121, 122, 124, 125, 126, 129, 131, 135, 136, 137

REST *Representational State Transfer* 88, 110, 118

RIA *Rich Internet Applications* 3, 35

SI *Service Interface* 73

SQL *Structured Query Language* 3, 8, 21, 25, 26, 28, 32, 36, 37, 55, 57, 70, 72, 103, 122, 137

SSL *Secure Sockets Layer* 121

SUT *System Under Test* 97, 100, 127

SaaS *Software As A Service* 55

TAFIM *Technical Architecture Framework for Information Management* 10

TCO *Total Cost of Ownership* 31

TCP *Transmission Control Protocol* 4

TOGAF *The Open Group Architecture Framework* 10

TPC *Transaction Processing Performance Council* 96, 128

TPC-W *TPC Benchmark W* 96, 97, 100, 114

ThickDB *Thick Database* 36, 37, 70, 71, 72, 125, 136

UI *User Interface* 31, 32, 73, 96, 102, 107, 110, 111

URI *Uniform Resource Identifier* 79, 80

URL *Uniform Resource Locator* 19, 42, 45, 47, 50, 52, 65, 67, 79, 80, 81, 83, 84, 85, 86, 89, 90, 101

WAR *Java Web Archive* 44

WIRT *Web Interaction Response Time* 129, 130

WoD *Window-on-Data* 36

XE *Express Edition* 21

XML *Extensible Markup Language* 73

XSS *Cross-Site Scripting* 26

YAFET *Yet Another Front End Technology* 35

Tabla de contenidos

Lista de siglas XII

1 Introducción 1

2 Estado del arte de arquitecturas centradas en el RDBMS 9

2.1 Definiciones y alcance 10

2.2 Arquitectura de aplicaciones web 12

2.3 Oracle Forms 17

2.4 Oracle Application Express 21

2.5 Trabajos relacionados con Oracle APEX 23

2.6 Trabajos con el enfoque database-centric 26

3 Análisis de Oracle APEX 39

3.1 Metodología 39

3.2 Vista lógica 40

3.3 Vista física 44

3.4 Vista de procesos 46

3.5 Vista de desarrollo 51

3.6 Consideraciones sobre el IDE 63

3.7 Vista de casos de uso 64

3.8 El caso del manejo de sesión 66

3.9 Dos lecciones del análisis de APEX 69

4 Propuesta de la arquitectura RDBMS-only 71

5 Prueba de concepto 76

5.1 El módulo mod_plpgsql 77

5.2 Manejo de estado 81

5.3 El prototipo de aplicación webpg 83

5.4 Los endpoints para HTTP GET y HTTP POST 88
5.5 Proceso de desarrollo y testing 91

6 Experimentos 95
6.1 El TPC Benchmark W 97
6.2 La aplicación TPCW 100
6.3 Capa de DL Code 104
6.4 Capa de BL Code 107
6.5 Capa de Interface Wrapper 109
6.6 Capa de UI Code 110

7 Análisis comparativo y evaluación del enfoque propuesto 115
7.1 Compatibilidad 117
7.2 Fiabilidad 119
7.3 Seguridad 121
7.4 Portabilidad 123
7.5 Mantenibilidad 124
7.6 Desempeño 126
7.7 Resumen del análisis 131

8 Conclusiones y trabajo a futuro 133

Bibliografía 139

Capítulo 1

Introducción

> [...] nuestros productos se convierten, en amplia medida, en independientes de sus artífices.
>
> *Búsqueda sin término*
> *Karl R. Popper*

En este trabajo se elabora la hipótesis de que una arquitectura de aplicaciones web físicamente contenida en un *Relational Database Management System* (RDBMS), a la cual se le ha llamado *arquitectura RDBMS-only*, es una alternativa válida, con ventajas y desventajas respecto de las bien establecidas arquitecturas de múltiples niveles.

Es relevante precisar que esta hipótesis no es inconsistente con la teoría bien establecida de que las arquitecturas de múltiples niveles constituyen *un* marco conceptual adecuado para el diseño de aplicaciones web, ya que éstas reportan muchas e importantes ventajas. La hipótesis sería inconsistente, sin embargo, con una teoría que afirmara que las arquitecturas de múltiples niveles constituyen *el único* marco conceptual adecuado para el diseño de aplicaciones web, sin dejar lugar para arquitecturas alternativas. Si bien esta última teoría no se ha afirmado de forma explícita, se presentarán algunos ejemplos que demuestran que en ocasiones se ha asumido de forma implícita, y asimismo se discutirá la poca literatura disponible que presente alternativas.

En otras palabras, este trabajo no pretende afirmar que una arquitectura *RDBMS-only* sea *mejor* que una arquitectura de múltiples niveles independientemente del contexto. El objetivo es más modesto. Se pretende

defender la validez de la alternativa describiéndola de forma detallada, mostrando el ejemplo de una tecnología real que sigue esta arquitectura, presentando el proceso y los resultados del desarrollo de un prototipo funcional y discutiendo ventajas y desventajas de la propuesta. Vale mencionar que en todo el presente trabajo ha de entenderse "funcional" en su acepción de *"eficazmente adecuado a sus fines"*, y nunca relacionado al concepto de programación o lenguaje funcional. Pese a la modestia del objetivo, se entiende que por un lado la tarea no es sencilla, porque supone la defensa de una estrategia que va a contracorriente de la tendencia general de mantener la lógica fuera del RDBMS, y por otro lado que aporta algún valor la descripción de la arquitectura propuesta, que no tiene antecedentes.

Típicamente, entre las decisiones tecnológicas que puede implicar el desarrollo de una aplicación web, se encuentran la elección de un lenguaje de programación de propósito general y de un *Database Management System* (DBMS), pero suele asumirse que la arquitectura básica de la aplicación será de múltiples niveles donde al menos el nivel de persistencia de los datos estará bien separado de los niveles de la lógica y la presentación [1]. Esta separación es bien clara porque suele no solo ser conceptual sino tecnológica, cuando el nivel de persistencia reside en un DBMS mientras que los niveles de la lógica y la presentación se programan en uno o más lenguajes de propósito general. Suscribiendo a la idea de que no hay ni habrá una bala de plata en la ingeniería de software [2], podemos conjeturar que las arquitecturas de múltiples niveles para las aplicaciones web, aunque es cierto que reportan muchos e importantes beneficios, simplemente podrían no ser la mejor solución en todos los casos.

En los últimos años se han llamado plataformas de aplicaciones de tipo *Low-Code* a un conjunto de soluciones con la característica común de permitir desarrollo de aplicaciones rápido, despliegue en un paso, y desarrollo y gestión utilizando un lenguaje declarativo de alto nivel. Esta clasificación, sin embargo, dice poco de la arquitectura subyacente a estas plataformas, la mayor parte de las cuales están disponibles como servicios de nube pública. En este caso, la decisión podría consistir en la elección de la solución de *Low-Code*, llamadas *Low-Code Application Platform* (LCAP) o *Low-Code Development Platform* (LCDP), y podríamos perder de vista las particularidades de la arquitectura [3, 4]. Algunas de estas plataformas *Low-Code*, tienen una arquitectura de múltiples niveles [5].

En cualquier caso, una aplicación con persistencia supone la existencia de

algún tipo de DBMS. No pocas veces ocurre que éste es un RDBMS, basado en el modelo relacional [6, 7, 8, 9, 10], con una interfaz para *Structured Query Language* (SQL) [11], y transacciones con las propiedades de Atomicidad, Consistencia, Aislamiento y Durabilidad, más conocidas por el acrónimo de los términos en inglés: *Atomicity, Consistency, Isolation, Durability* (ACID) [12]. Si bien el poder expresivo del lenguaje SQL, con el agregado de la recursión, ha sobrepasado al del álgebra relacional y la lógica de primer orden, tanto en su especificación como en las implementaciones de los principales RDBMSs, no es fácil definir con claridad los límites de este nuevo poder expresivo, y más allá de la posibilidad de resolver ciertos problemas utilizando un lenguaje declarativo como SQL, muchos problemas se resuelven con mayor facilidad utilizando un lenguaje de programación de propósito general, sea procedural u orientado a objetos, que permita el uso de secuencia de instrucciones, estructuras de control, variables y manejo de errores [13]. Los principales RDBMSs de la industria tienen al menos un *Database Programming Language* (DBPL) incorporado, como PL/SQL en el caso de Oracle, Transact-SQL en el caso de SQL Server, SQL/PL en el caso de DB2 o PL/pgSQL en el caso de PostgreSQL. De esta forma, los RDBMSs permiten la utilización de uno o más lenguajes de programación imperativa dentro del propio RDBMS, con lo cual es posible mantener código imperativo encapsulado en procedimientos, que suelen llamarse *stored procedures*. Utilizando una combinación de SQL y *stored procedures* podría programarse toda la lógica de una aplicación, y esto de hecho es una alternativa no solo teóricamente válida sino ampliamente utilizada en la industria, como sucede con las aplicaciones desarrolladas con *Oracle Fussion Middleware* [14]. De esta forma, es posible mantener tanto la persistencia como la lógica dentro del RDBMS, y esto permite soportar aplicaciones intensivas en el uso de datos con un *framework* único, simple en su base conceptual y libre de desajustes técnicos [15].

Requerimientos específicos, como la utilización de *Rich Internet Applications* (RIA) [16] mediante acceso web o *Machine Learning* (ML) usualmente determinan que se necesite recurrir a alguna tecnología externa al RDBMS. Sin embargo, por un lado la industria está invirtiendo en la integración de algoritmos de ML y aceleradores de hardware especializados dentro de los RDBMS [17]. Por otro lado, el RDBMS a través de SQL y sus *stored procedures*, puede generar las partes estáticas y dinámicas de las páginas de RIA. Si la lógica para generar las interfaces se puede incluir en el RDBMS,

tenemos que las capas clásicas de una aplicación de tres capas (persistencia, lógica y presentación) [1] podrían residir todas en el nivel del RDBMS, y en este extremo tendríamos la arquitectura que en este trabajo se ha denominado *RDBMS-only*.

Existe una diferencia fundamental entre las solicitudes de los clientes a través de un *browser* utilizando el protocolo *Hypertext Transfer Protocol* (HTTP), que no es orientado a conexión, y las sentencias que se ejecutan en el RDBMS que típicamente atiende por una *Application Programming Interface* (API) propietaria [1] sobre protocolo *Transmission Control Protocol* (TCP), orientado a conexión. Lo que se necesita es entonces un componente ubicado entre el *browser* de los clientes y el RDBMS, que pueda resolver los problemas derivados de esa naturaleza diferente de los protocolos. Este componente debería atender los *HTTP Requests*, comunicarse con el RDBMS (sea a través de la API estándar de un driver o directamente a través de la API propietaria) y retornar el *HTTP Response* adecuado. Para lograr esto y mantenerlo acotado, este componente podría delegar nuevamente al RDBMS la tarea de mantener cierto estado de las sesiones; con lo cual se transforma en un componente esencialmente tecnológico, y puede ser visto como una extensión más del RDBMS: un *web listener*.

Todas estas propuestas teóricas no necesitan un análisis de factibilidad, ya que existe en la industria un producto funcional y de escala empresarial llamado *Oracle Application Express* (APEX) que sigue estos lineamientos generales, que de hecho fueron descritos de esta forma a partir de la generalización de la propia arquitectura del producto. El caso de APEX parece ser el único en la industria que utiliza esta arquitectura, y ésta no ha sido analizada en la literatura, ni tiene un nombre. APEX tiene una arquitectura física muy simple, similar a la que en este trabajo se denomina *RDBMS-only*, ya que las aplicaciones desarrolladas con este entorno (y el entorno mismo) tienen tanto los datos, como la lógica de dominio y la interfaz, todo en el RDBMS, y fuera de éste se valen solo de un pequeño componente tecnológico, haciendo las veces de un *web listener*, que se ubica entre el *browser* del cliente y el RDBMS, y que se encarga de traducir entre HTTP y el lenguaje procedural del RDBMS. En el caso de APEX este pequeño componente puede ser un módulo

[1] Vale aclarar que está justificado decir que siempre hay una API propietaria, ya que si bien existen formas de acceso estándar como *Java Database Connectivity* (JDBC), son drivers que ofrecen una API estándar pero internamente utilizan una API propietaria.

de Apache, una aplicación Java Standalone, una aplicación Java web o incluso una funcionalidad añadida al propio RDBMS, y en realidad estas alternativas no cambian la estructura de la arquitectura general [2]. Ya que en la industria no han aparecido aún otros productos con esta misma arquitectura, y en la academia tampoco parece haber despertado interés, parece válida la pregunta de por qué este producto sigue teniendo una "arquitectura única" [18]. Desde el punto de vista teórico, aún reconociendo que va a contracorriente de la tendencia general, es una arquitectura de aplicaciones válida y que merecería describirse con detalle para identificar claramente sus ventajas y desventajas, su forma de funcionamiento, los desafíos tecnológicos que presenta de cara a la implementación y los casos ideales donde esta arquitectura se mostraría más conveniente. La referida "contracorriente" no debe entenderse solamente respecto de una tendencia a utilizar múltiples niveles, sino fundamentalmente respecto de la tendencia a no utilizar el DBPL del RDBMS para mantener la lógica de la aplicación.

Vale aclarar que la defensa de la validez de esta arquitectura no representa de ninguna manera una crítica al principio de separación de intereses, ya que pueden seguir existiendo varias capas conceptualmente separadas con responsabilidades diferentes bien definidas, aún cuando físicamente se encuentren todas dentro de un RDBMS. Sin embargo, el hecho de que físicamente toda una solución se encuentre encapsulada en un RDBMS supone problemas diferentes a los que se deben enfrentar con otras arquitecturas, y motiva el análisis de estos problemas en la búsqueda de alternativas de solución.

En este punto, es necesario realizar una aclaración terminológica. Distinguiremos entre los conceptos de *capa* (traducción de *"layer"*) y *nivel* (traducción de *"tier"*). A nivel conceptual, típicamente los sistemas de información se diseñan considerando tres *capas*:

- *capa de presentación*
- *capa de lógica de aplicación* o *capa de lógica de negocio*, y
- *capa de gestión de recursos* o *capa de acceso a datos.*

La *capa de presentación* es responsable de administrar la interfaz con el cliente, siendo cliente cualquier usuario o programa que quiera operar con el sistema. La

[2] De estas alternativas el módulo de Apache (mod_plsql) y la funcionalidad añadida al propio RDBMS (Embedded PL/SQL Gateway) están deprecadas desde APEX 20.1, lo cual es irrelevante. El punto es que mientras estuvieron soportadas demostraron su viabilidad.

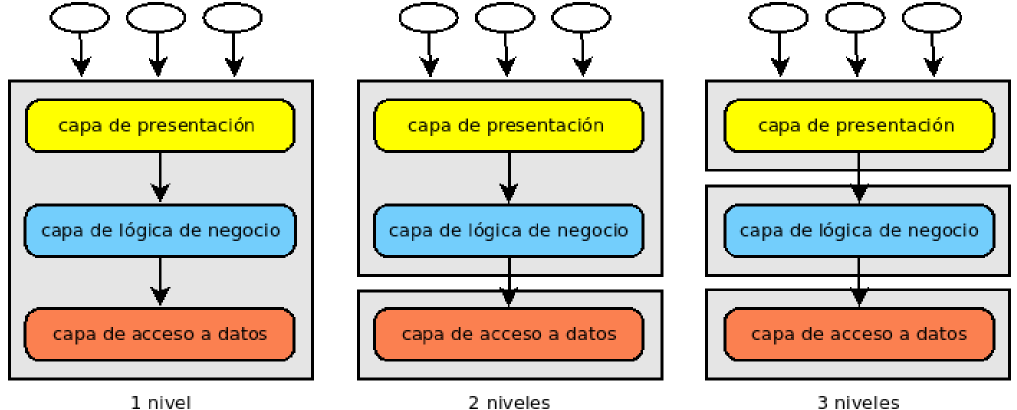


Figura 1.1: Sistemas de uno, dos y tres niveles

capa de lógica del negocio establece qué operaciones se pueden ejecutar sobre el sistema y cómo se desarrollarán, se encarga de asegurar el cumplimiento de las reglas de negocio y de ejecutar los procesos de negocio. Puede implementarse mediante restricciones, procesos de negocio o programas. La *capa de acceso a datos* ejecuta funciones de base de datos como recuperación, inserción, eliminación y modificación de registros [19]. A nivel de implementación, estas *capas* se pueden combinar y distribuir en uno o múltiples *niveles* (lo que usualmente significa servidores) separados. Muchas aplicaciones web modernas utilizan una arquitectura de tres *niveles* que se denominan comúnmente *nivel de cliente* (traducción de *"client-tier"*), *nivel medio* (traducción de *"middle-tier"*) y *nivel de persistencia* o *nivel de datos* (traducción de *"data-tier"*).

La Figura 1.1 presenta los sistemas más clásicos de uno, dos y tres niveles. Los sistemas de un nivel algunas veces se describen como "monolíticos", término cuyas acepciones se discutirán más adelante. Los sistemas de dos niveles se presentan generalmente como "cliente-servidor", concepto que también puede tener más de una acepción y sobre el que volveremos. Los sistemas de tres niveles típicamente dedican un nivel a cada capa. En este esquema la *capa de lógica de negocio* coincide con un nivel que está "en el medio", entre el *nivel de cliente* y el *nivel de datos*, lo que constituye un *nivel medio* en el sentido físico. Esta situación puede explicar por qué los términos "capa de lógica de negocio", "capa intermedia" y "nivel intermedio" a veces se utilizan como sinónimos [20].

A su vez, para hacer interoperar varios sistemas, se puede agregar otra capa con lógica de integración, usualmente denominada *"middleware"*, y como esta capa podría a su vez subdividirse en subcapas e incluso ser en sí misma

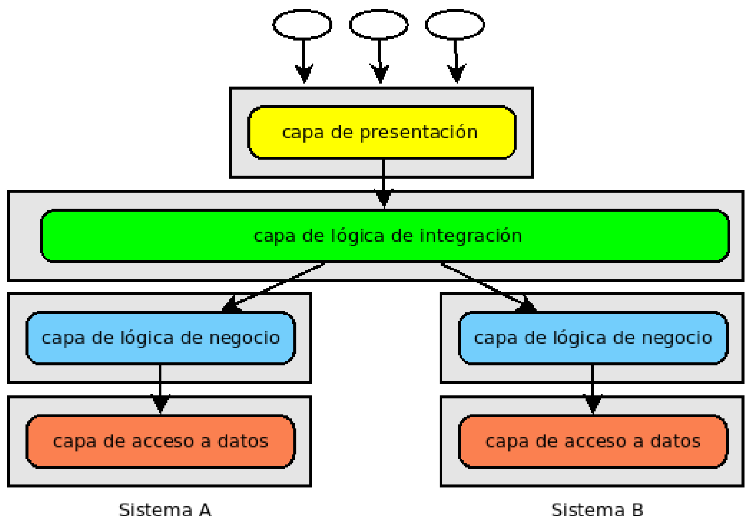


Figura 1.2: *Middleware* y sistemas de N niveles

un sistema con más de un nivel, este tipo de sistemas se denominan de forma genérica de N niveles [20]. La Figura 1.2 presenta la arquitectura de un sistema de N niveles que incluye un *middleware*.

En líneas generales, el número de niveles de muchos sistemas ha crecido en el tiempo, lo que ofrece flexibilidad al costo de una mayor complejidad de la arquitectura y una penalización de desempeño [20].

La arquitectura *RDBMS-only* que se presenta en el Capítulo 4 tiene un solo *nivel*, pero de ninguna manera tiene una sola *capa*. Las tres *capas* conceptuales existen, por lo que se puede respetar la separación de intereses y no es válido suponer que se trata de una arquitectura monolítica, si por este concepto entendemos la carencia de modularidad. Sin embargo, deberemos admitir que se trata de una arquitectura monolítica entendida como "una aplicación con una única base de código/repositorio grande que ofrece decenas o cientos de servicios utilizando diferentes interfaces" [21] .

Puede argumentarse que los DBMSs no fueron originalmente concebidos para el desarrollo de aplicaciones, pero los productos humanos "se convierten, en amplia medida, en independientes de sus artífices", evolucionan con vida propia, y son fértiles en el sentido que podemos aprender de ellos y darles usos que no estaban presentes en las intenciones de sus creadores [22]. Esto ya ha sucedido con los DBMSs, y también se ha predicho que podrían especializarse en los próximos años para cumplir propósitos específicos, liberándose de todo el overhead propio de una solución genérica [23]. Un entorno *RDBMS-only* podría representar una de estas direcciones de especialización. Algún caso de

uso posible de aplicaciones con arquitectura *RDBMS-only*, como un *Content Management System* (CMS) o un sistema de *e-commerce*, podría beneficiarse de un rediseño completo del RDBMS sobre la premisa de incluir lo necesario para la solución autocontenida pero no características extra de propósito general que no siendo útiles generaran overhead (e.g. simplificación del SQL, del optimizador, de los *isolation levels*, de los lenguajes procedurales, de los tipos de datos).

En este trabajo se intenta aportar un análisis de la arquitectura *RDBMS-only* tanto desde la perspectiva teórica como a partir de algunas lecciones aprendidas durante las pruebas de implementación y experimentos con esta arquitectura. El Capítulo 2 presenta un análisis del estado del arte sobre las arquitecturas de aplicaciones web centradas en un RDBMS, mientras que el Capítulo 3 se ocupa de analizar en detalle la arquitectura de APEX. En el Capítulo 4 se propone una arquitectura *RDBMS-only* a nivel teórico, que se utiliza como guía en el Capítulo 5 donde se realiza una implementación parcial como prueba de concepto llegando a un prototipo funcional que podría extenderse. En el Capítulo 6 se experimenta con la implementación de dos aplicaciones en paralelo, una con Java y otra con el prototipo del Capítulo 5. En el Capítulo 7 se realiza un análisis comparativo entre la arquitectura *RDBMS-only* y otras arquitecturas, con especial énfasis en el desempeño. Finalmente, el Capítulo 8 presenta las conclusiones y se indican algunos problemas futuros a abordar para continuar con esta línea de trabajo.

Algunos de los resultados de esta tesis se presentaron en forma de artículo, y fueron aceptados, en la edición XLVII de la Conferencia Latinoamericana de Informática (CLEI). El nombre del artículo es *An RDBMS-Only Architecture for Web Applications* [24].

Capítulo 2

Estado del arte de arquitecturas centradas en el RDBMS

No es tierra, sino un animal,
donde preparamos nuestra fiesta.
Es un pez marino de los mayores.

Viaje de San Borondón
Benedeit

Este capítulo pretende ofrecer un análisis del estado del arte sobre las arquitecturas de aplicaciones web centradas en un RDBMS. La Sección 2.1 presenta la definición de arquitectura de software utilizada en este trabajo, y a qué clase de software se limitará el análisis. La Sección 2.2 describe la evolución y las tendencias generales de las arquitecturas de aplicaciones web, mientras que las secciones 2.3 y 2.4 presentan respectivamente las arquitecturas de *Oracle Forms* y APEX, dos productos que no solo siguen una arquitectura diferente a las mencionadas anteriormente, sino que representan una evolución en una dirección diferente a la de la tendencia general. En la Sección 2.5 se presentan trabajos directamente relacionados con APEX, y finalmente en la Sección 2.6 se presentan trabajos sobre arquitecturas centradas en el RDBMS en general.

2.1. Definiciones y alcance

Cuando el software es inherentemente complejo, una forma conveniente de abordarlo es con un diseño modular, utilizando abstracción u ocultamiento de la información de implementación, y complementando esta abstracción con una guía modular en forma de documento estructurado [25]. Este interés por la estructura y la documentación de un diseño modular derivó en la disciplina de la arquitectura de software [26]. Si bien no hay una definición universalmente aceptada, generalmente por arquitectura se entiende la descripción de un conjunto de componentes y sus relaciones, así como los principios que gobiernan su diseño y evolución en el tiempo. Existen varios modelos, o *frameworks*, que prescriben qué describir de una arquitectura y cómo hacerlo. Algunos de estos *frameworks* han sido desarrollados dentro de una empresa como IBM [27, 28], otros por organizaciones estatales como el Departamento de Defensa de los Estados Unidos [29], y otros por consorcios de organizaciones como el Open Group [30]. Los diferentes *frameworks* pueden tener diferencias de objetivos, enfoques y niveles de abstracción, han surgido en diferentes momentos, algunos han sido influenciados por otros y algunos de ellos mantienen una evolución hasta hoy. A pesar de las diferencias, lo constante es la intuición que Perry y Wolf desarrollaron sobre la arquitectura de software, basándose en las disciplinas de la arquitectura de hardware, la arquitectura de redes y la clásica arquitectura de edificios [31]. Sobre ésta última, si bien los autores reconocen que los edificios son muy diferentes al software, mencionan que hay ideas sugestivas y aplicables a la arquitectura de software, como las múltiples vistas, los estilos arquitectónicos, y las relaciones entre el estilo y la ingeniería o los materiales. El constructor, el arquitecto, el diseñador de interiores y el electricista tienen todos diferentes vistas del edificio, y todas ellas son válidas, útiles y necesarias. Estos autores pronosticaban en 1992 que la década del 90 sería la de la arquitectura de software, y efectivamente se propusieron varios *frameworks* en esa década, algunos de los cuales aún hoy se utilizan. Un antecedente temprano es el *Zachman Framework* presentado en 1987 y extendido en 1992 [27, 28], y posteriormente se propusieron otros como *Enterprise Architecture Planning* (EAP) en 1993 [32], *The Open Group Architecture Framework* (TOGAF) en 1995 [30], *Technical Architecture Framework for Information Management* (TAFIM) en 1996 [26], *Command, Control, Communications, Computers,*

Intelligence, Surveillance and Reconnaissance (C4ISR) en 1997 [29], o *Federal Enterprise Architecture Framework* (FEAF) en 1999 [33]. Esta enumeración no es taxativa, y solo tiene el espíritu de demostrar el interés por la arquitectura existente en esa década. En general, esos *frameworks* pretenden abarcar una *"Enterprise Architecture"*, donde por *"Enterprise"* se entiende que la arquitectura incluye otras dimensiones pertenecientes al dominio de la organización y que trascienden el software, como los datos o los procesos de negocio. En ocasiones, sin embargo, se puede pretender describir y analizar únicamente el software, y en ese caso puede ser suficiente un modelo como el de 4+1 vistas presentado en 1995 [34].

El modelo 4+1 organiza la descripción de una arquitectura de software utilizando cinco vistas concurrentes, cada una de las cuales aborda un conjunto específico de intereses. Este modelo captura las decisiones de diseño en cuatro de las vistas y utiliza la quinta para ilustrarlas y validarlas. La vista lógica describe el modelo de diseño de objetos, cuando se utilizan metodologías o lenguajes orientados a objetos, pero puede utilizarse otra forma de vista lógica, como un *Modelo Entidad-Relación* (MER) cuando la aplicación es *"data-driven"*. La vista de procesos describe los aspectos del diseño de la concurrencia y sincronización. La vista física describe la relación del software con el hardware (y actualmente, podría acotarse, con los *hosts* sean éstos físicos o virtuales) reflejando su naturaleza distribuida. La vista de desarrollo describe la organización estática del software en su ambiente de desarrollo. Se puede ilustrar y complementar la información de estas cuatro vistas, con unos pocos casos de uso, que conforman la quinta vista. En este trabajo, utilizaremos este modelo de 4+1 vistas para describir y comparar arquitecturas de software, ya que dejaremos fuera de alcance cualquier consideración que exceda al software en sí mismo, como las que involucran a la arquitectura de datos y procesos de negocio.

El alcance de este trabajo quedará limitado a una clase muy específica de software: aplicaciones web, de gestión y transaccionales. Asumiremos pues las siguientes definiciones en lo que sigue:

Definición 2.1 *Por aplicación web se entiende cualquier software que pueda ser accedido por un usuario a través de un web browser, utilizando HTTP.*

De acuerdo a la Definición 2.1, cualquier sitio informativo como los de estándares publicados, documentación de productos de software, o el de la

Real Academia Española (RAE), son ejemplos de aplicaciones web.

Definición 2.2 *Por aplicación de gestión se entiende cualquier aplicación que interviene en un sistema de información que pretende gestionar un conjunto de procesos sobre un dominio definido, donde los datos son relevantes al punto que diferentes tipos de datos pueden necesitar ser tratados de forma diferente, y se persisten en una base de datos por medio de un DBMS.*

De acuerdo a la Definición 2.2, un sistema de gestión de expedientes es un ejemplo de aplicación de gestión.

Definición 2.3 *Por aplicación transaccional se entiende cualquier aplicación que hace uso de las propiedades ACID [12] de las transacciones, propiedades que tradicionalmente han sido ofrecidas por los RDBMS.*

De acuerdo a la Definición 2.3, son ejemplos de aplicaciones transaccionales los sistemas de *Enterprise Resource Planning* (ERP), *Customer Relationship Management* (CRM) o *e-commerce*.

2.2. Arquitectura de aplicaciones web

Las aplicaciones web son accedidas por un *web browser*, que utiliza el protocolo HTTP para conectarse con un *web server*, quien le entrega los recursos solicitados, como páginas web, imágenes o código para ser ejecutado en el *web browser*. En este sentido, una aplicación web se puede ver como una aplicación cliente-servidor donde el *web browser* es el cliente mientras el *web server* es el servidor, como se observa en la Figura 2.1.

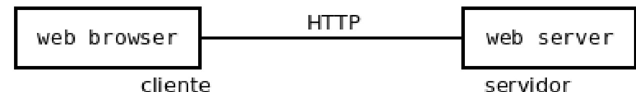


Figura 2.1: Arquitectura de una aplicación web como cliente-servidor

Este modelo simple podría ser suficiente para un sitio web estático, pero una aplicación web de gestión, para ser llamada como tal, debe ofrecer un servicio interactivo y generar contenido dinámico [35]. En cualquier caso la vista es incompleta pero válida en el caso general, ya que el *web server* es

la fachada donde atiende la aplicación, mientras alguna otra tecnología debe encargarse de ofrecer el servicio interactivo y generar contenido dinámico. Las aplicaciones web de gestión y transaccionales, que han necesitado persistencia con las propiedades ACID de las transacciones, utilizan usualmente una base de datos gestionada por un RDBMS. Y entre el *web server*, que sabe entregar contenido estático, y el RDBMS, debe existir algún programa que se integre con el *web server* por un lado y con el RDBMS por otro, y ejecute consultas y sentencias de manipulación en la base de datos. Esta extensión se presenta en la Figura 2.2, y el "programa" es una entidad conceptual que desde el punto de vista lógico se encuentra entre el *web server* y el RDBMS, aunque desde el punto de vista físico podría residir en un componente independiente, estar integrado en el *web server* o en el RDBMS, o incluso estar distribuido entre varios componentes que podrían incluir o no el *web server* y el RDBMS.

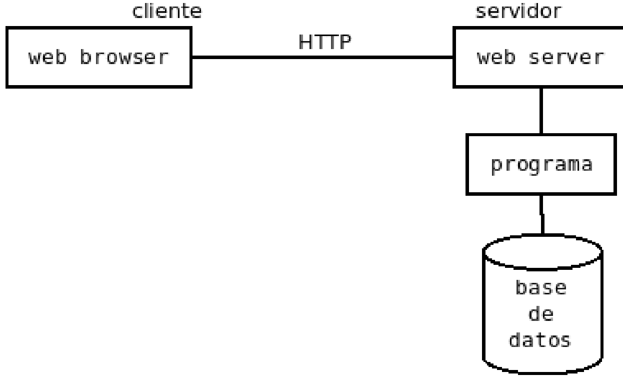


Figura 2.2: Arquitectura genérica de una aplicación web con persistencia

Uno de los primeros acercamientos al problema fue una propuesta surgida de la industria que se formalizó en el protocolo *Common Gateway Interface* (CGI) en 1993 [36]. Este protocolo especifica la forma en que un *web server* puede interactuar con programas externos, llamados *CGI scripts*, así como la forma en que éstos retornan al *web server* las páginas dinámicas. El paradigma de la *Object-Oriented Programming* (OOP) se afianzó en la industria desde la década del 80 especialmente con la adopción de lenguajes de programación de propósito general orientados a objetos, y estaba en pleno auge en la década del 90 cuando surgió la web [37]. Debido a la funcionalidad limitada de CGI, se propuso que los objetos debían salir "al rescate" [38], y de hecho las propuestas para aplicaciones web posteriores a CGI han seguido en general la tendencia

de incorporar la orientación a objetos donde estaba más madura, es decir, en los lenguajes de programación de propósito general, y no en las bases de datos donde los *Object-Oriented Database Management System* (OODBMS), no estaban maduros y fueron escasamente adoptados en la industria [39]. Esto ha implicado una tendencia a agregar funcionalidades y complejizar el componente del "programa" de la Figura 2.2 y generalmente utilizando un lenguaje orientado a objetos, al punto que en algunos trabajos se presenta la programación imperativa como una etapa superada en la historia de la computación [37, 40]. Como puede verse en la Tabla 2.1, se postulaba que a partir de la década de 1990 los lenguajes serían Orientados a Objetos.

Decade	Batch 1960s	Time-Sharing 1970s	Desktop 1980s	Network 1990s
Operation	Process	Edit	Layout	Orchestrate
Interconnect	Peripherals	Terminals	Desktops	Palmtops
Applications	Custom	Standard	Generic	Components
Languages	Cobol, Fortan	PL/I, Basic	Pascal, C	Object-Oriented

Tabla 2.1: "The Four Paradigms of Computing", de Tesler, 1991 [40]

A los efectos de interoperar con un RDBMS los lenguajes procedurales tienen ventajas respecto de los lenguajes orientados a objetos [41], y también hay trabajos que postulan que la programación orientada a objetos, lejos de ser más natural que la programación imperativa, es menos intuitiva, porque los seres humanos estamos predispuestos a pensar de forma procedural y jerárquica [42].

En este contexto, se propusieron las arquitecturas de tres niveles, con los niveles de *front-end clients*, *application servers* y *database server*, siendo en el nivel de los *application servers* donde se mantiene (utilizando un lenguage orientado a objetos) toda la lógica de la aplicación [43]. Con esta división se tiene que en el nivel de los *front-end clients* ejecuta un *web browser* y eventualmente algún *plug-in* o código que se ejecuta interpretado por el propio *web browser*, en el nivel de los *application servers* ejecuta el *web server* y alguna otra tecnología de intérprete de lenguajes o contenedores de aplicación, mientras que en el nivel del *database server* típicamente ejecuta un RDBMS. Es interesante destacar que todas las tecnologías de aplicaciones web con persistencia siguen esta arquitectura genérica de la Figura 2.2, y a partir de

ella pueden especializarla, como se muestra en la Figura 2.3 para el caso de *Jakarta Enterprise Edition*. En el ejemplo de la Figura 2.3 lo que entendemos por "programa" es un conjunto de componentes de software, pero aún puede concebirse como un único componente (compuesto) que está en el medio entre la comunicación por HTTP con el cliente y la comunicación por un protocolo propietario del DBMS para resolver la persistencia.

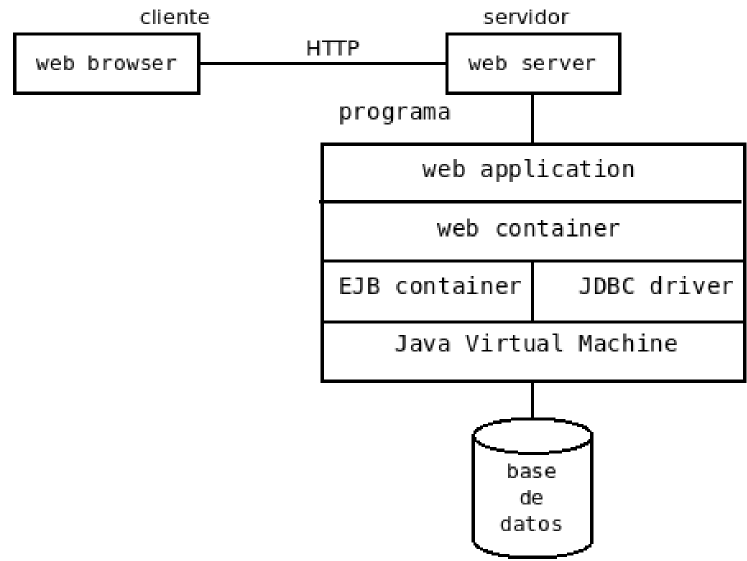


Figura 2.3: Arquitectura de una aplicación *Jakarta Enterprise Edition*

En cuanto a las bases de datos, si bien hubo propuestas para incorporar la Orientación a Objetos en los DBMSs, éstas tuvieron algunos problemas y fueron escasamente adoptadas en la industria [39]. La tendencia general fue resolver el problema de la interoperabilidad con un componente dedicado a implementar el *Object-Relational Mapping* (ORM). Si bien actualmente ha habido mucho interés por los sistemas NoSQL, y algunos de ellos ofrecen transacciones con las propiedades ACID, hay espacio para ambos sistemas en el mercado, los RDBMSs aún tienen su nicho en las aplicaciones internas de gestión y su demanda no desaparecerá en el corto plazo [44].

Con esto se consolidó el enfoque que proponía centrar la atención en las capas del dominio y la presentación, delegando la responsabilidad de resolver la mayor parte de los problemas de la capa de persistencia, como problemas tecnológicos que son, a una tecnología de ORM. Esta tendencia ha sido justificada, ya que la POO permite realizar diseños modulares basados en

objetos que “hablan” entre sí y que ocultan su implementación ofreciendo una interfaz hacia el exterior; lo que contribuye, a su vez, a diseñar componentes reusables, con alta consistencia interna y bajo acoplamiento [45].

Las crecientes demandas de construir aplicaciones distribuidas complejas manteniendo un bajo acoplamiento, han tenido como respuesta la propuesta de diseñar las aplicaciones utilizando *Web Services* y capas de *Middleware*, entre las bases de datos y los browsers de los clientes [46]. En este modelo cada capa utiliza los servicios de la capa subyacente, y proporciona servicios más especializados. Se ha propuesto también una arquitectura llamada *Distributed Object Computing* (DOC) *Middleware* [47], un “paradigma” [1] que soporta un comportamiento flexible y adaptable, basado en la agregación de interacciones simples a través de las capas del *DOC Middleware*, como puede observarse en la Figura 2.4 [48]. En esta propuesta concurren dos tendencias que frecuentemente se ven juntas: la orientación a objetos y la distribución en múltiples niveles físicos, que están presentes en la *Object Management Architecture* (OMA) del *Object Management Group* (OMG). Asimismo, postulaba que Java era un lenguaje orientado a objetos que eliminaba las ineficiencias de CGI [47].

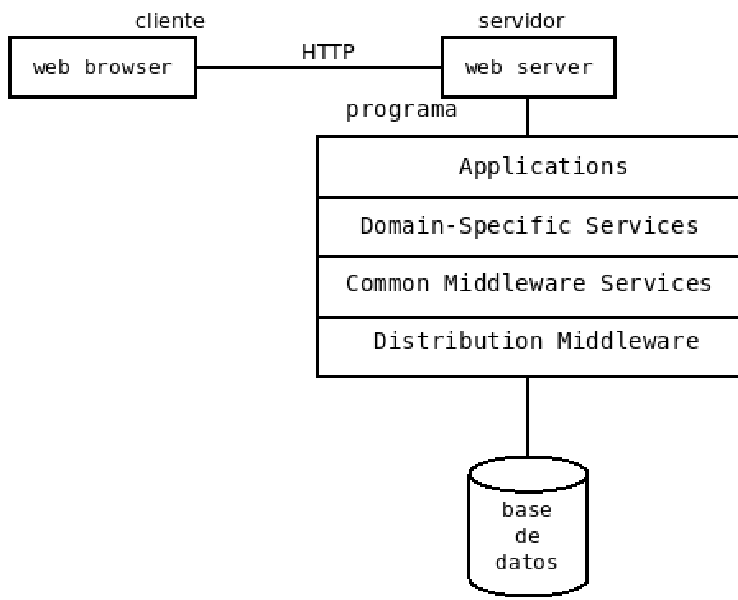


Figura 2.4: *Distributed Object Computing Middleware*

En cuanto a la distribución física, se pasó de tener tres niveles a tener

[1] La inclusión de este término no es gratuita, el trabajo donde se presentaba esta arquitectura se tituló *The distributed object computing paradigm: concepts and applications.*

múltiples niveles, debido a esta especialización del nivel de los *application servers*.

En los últimos años esta evolución ha seguido su curso, siempre modificando, ampliando y complejizando estas capas y niveles de *Middleware* entre el *web browser* y el RDBMS. Dos excepciones a esta regla son los casos de *Oracle Forms* y APEX, que se pueden catalogar como arquitecturas *database-centric* o *database-driven* [2], y se describen en las secciones 2.3 y 2.4 respectivamente.

2.3. Oracle Forms

Oracle Forms (en adelante, Forms) es un producto que ofrece interfaces para interoperar con un RDBMS Oracle y se puede considerar un entorno de *Rapid Application Development* (RAD) o *Low-Code*. El precursor de Forms se llamaba *Interactive Application Facility* (IAF) y data de 1979 con Oracle release 2, y era una tecnología de servidor con interfaz de caracteres, diseñada para ser ejecutada a través de terminales. El producto evolucionó a SQL*Forms en 1986 con Oracle release 5, versión en la que se agregó la posibilidad de conectar clientes a través de una red, convirtiéndose en una tecnología cliente-servidor, donde los clientes ejecutaban los Forms localmente a través de un intérprete llamado *Forms Runtime Module*, y se conectaban con el RDBMS Oracle a través de un protocolo propietario, llamado *Oracle Call Interface* (OCI). El lenguaje PL/SQL se incorporó en 1988 en *Oracle Forms* 3 con Oracle release 6. PL/SQL es la propuesta de Oracle enmarcada en la tendencia general de los RDBMSs de incorporar como extensión algún tipo de DBPL, que permite almacenar y ejecutar código en lenguajes de programación imperativa inspirados en Pascal/R [49, 50]. Esta incorporación es un hito importante, junto con la incorporación de *stored procedures* en 1992 con Oracle release 7, lo que permitió la construcción de software modular manteniendo la lógica de la aplicación en la propia base de datos en forma de *stored procedures*.

[2] La semántica de las expresiones *databse-centric* y *database-driven* se discuten en el primer párrafo de la Sección 2.6

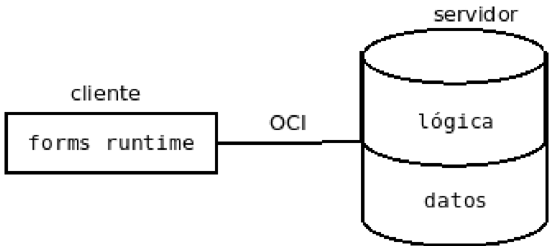


Figura 2.5: Arquitectura de *Oracle Forms* cliente-servidor

Oracle Forms 4.0, en 1993, fue la primera versión con una *Graphical User Interface* (GUI), que se podía ejecutar en Windows 3 y Unix Motif. La arquitectura era cliente-servidor, denominada por Oracle *SmartClient Architecture* [51]. Se presenta una vista lógica de esta arquitectura en la Figura 2.6.

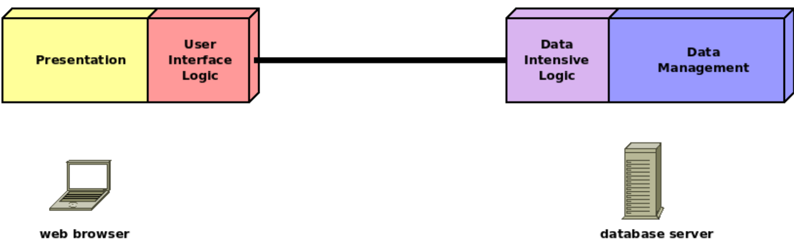


Figura 2.6: Vista lógica de la arquitectura de Forms Cliente-Servidor. Producción propia con base en [51]

Oracle Forms 6i (donde la i viene de internet), en 2000, incluyó la posibilidad de que el cliente accediera a través de un *web browser*, conectándose a un *application server*. Este último incluía un *Forms Server*, un componente diseñado para solucionar el problema de adaptar *Oracle Forms* a una arquitectura física de tres niveles, y que permitía que los forms se descargaran y ejecutaran a través de un *Java Applet*. Con este esquema, ilustrado en la Figura 2.7, el inicio de la ejecución de una aplicación Forms ocurre a través de un *web browser* y por HTTP, pero a partir de la descarga del *Java Applet*, la ejecución de éste ocurre dentro de un *plugin* del *web browser* y por un protocolo propietario llamado *Network Computing Architecture* (NCA) atendido por un proceso de servidor llamado *Forms Runtime Engine* [51].

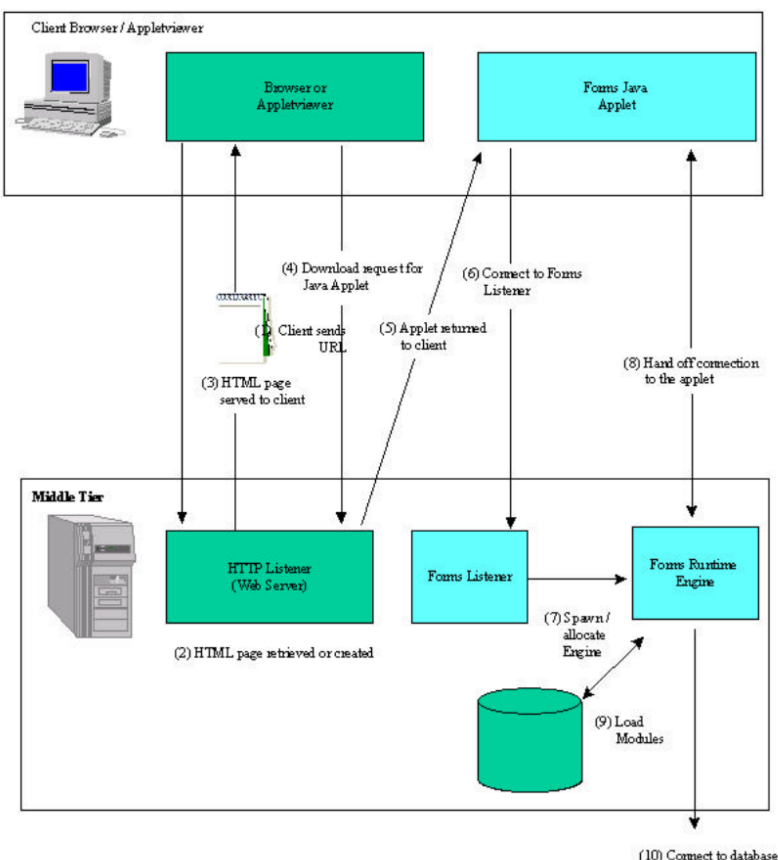


Figura 2.7: Vista de procesos de la arquitectura de *Forms Web* [52]

La secuencia de pasos en la Figura 2.7 se describe a continuación:

1. El usuario accede a la *Uniform Resource Locator* (URL) de una página *HyperText Markup Language* (HTML) que indica que se debe ejecutar una aplicación de Forms.
2. La página HTML se descarga en el navegador web. Si es necesario, el cliente también descargará el archivo Java que contiene el *Forms Applet*. Se instanciará el *Forms Applet* y se utilizarán los parámetros de la página HTML para determinar qué aplicación de Forms se deberá ejecutar.
3. El *Forms Applet* envía un *request* al *Form Listener* (que reside en un puerto específico del host desde el que se descargó el *Forms Applet*).
4. El *Forms Listener* se pone en contacto con la *Forms Runtime Engine* y se conecta a un proceso del *Forms Server*.
5. El *Forms Listener* establece una conexión con la *Forms Runtime Engine* y envía la información de conexión al *Forms Applet*.

6. El *Forms Applet* establece una conexión directa con la *Forms Runtime Engine*
7. El *Forms Applet* y la *Forms Runtime Engine* se comunican directamente, liberando al *Forms Listener* para aceptar solicitudes de otros usuarios. El *Forms Applet* despliega la interfaz de usuario de la aplicación en el *web browser* del usuario
8. La aplicación que se ejecuta en la *Forms Runtime Engine* se comunica directamente con la base de datos

Formalmente, podríamos decir que esta arquitectura no cumple con los requisitos de una aplicación web que establecimos en la Sección 2.1, ya que el protocolo utilizado solo es HTTP para la descarga del *Java Applet*. En cualquier caso, este camino adoptado por Oracle propone una arquitectura diferente a la clásica arquitectura de tres capas con presentación, lógica y persistencia separadas tecnológicamente. En esta propuesta la presentación se ejecuta del lado del cliente a través de un *Java Applet*, cierta lógica relacionada a la interfaz de usuario se ejecuta en el *Application Server*, mientras que la lógica que accede intensivamente a los datos se ejecuta en el RDBMS donde se mantiene en forma de *stored procedures* [51]. La Figura 2.8 presenta este esquema.

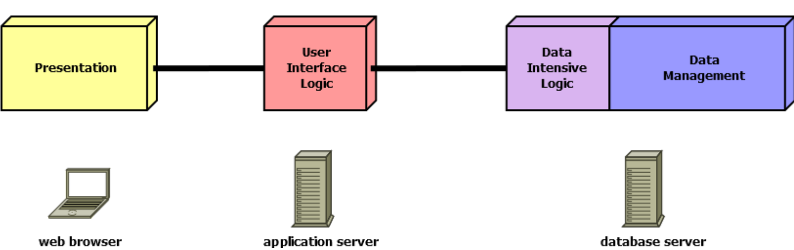


Figura 2.8: Vista lógica de la arquitectura de *Forms Web*. Producción propia con base en [51]

La ejecución de los Forms en un *web browser* ha ido quedando limitada por la tendencia de los *web browsers* a dejar de soportar *plugins*, y la respuesta de Oracle con Forms 12c, en 2015, fue agregar la posibilidad de utilizar Java Web Start, así como continuar explorando opciones para el despliegue de los Forms en el cliente [53]. El esquema de la Figura 2.8 sigue siendo válido, solo cambiando *web browser* por una *Java Virtual Machine* (JVM) local.

2.4. Oracle Application Express

APEX es un entorno ofrecido por Oracle, tanto para desarrollar aplicaciones desde cero, como para migrar aplicaciones desde *Oracle Forms*, o para desarrollar extensiones en otros productos como Oracle e-Business Suite. Esta tecnología no tiene costo adicional para quienes ya dispongan de un RDBMS Oracle, y puede obtenerse de forma gratuita junto con *Express Edition* (XE). Vale mencionar que XE tiene limitaciones en términos de recursos y no es una edición recomendada para un entorno productivo.

La experimentación con estas tecnologías comenzó a fines de la década del 90, e internamente hubo varios proyectos y cambios de nombre, incluyendo PL/SQL Web Toolkit, Web DB, Flows, Oracle Platform, Project Marvel y HTML DB. Formalmente se puede catalogar como una integración Web-DBPL que utiliza una extensión del DBPL [54]. La primera versión del *PL/SQL Web Toolkit* se liberó en 1999, en 2004 se liberó la primera versión del nuevo proyecto HTML DB, y en 2006 se renombró como APEX.

Esta solución está completamente contenida en una base de datos Oracle, y se compone únicamente de datos en tablas y código PL/SQL, que conforman un repositorio de metadata que mantiene la definición de las aplicaciones, y un motor, llamado *Application Express engine*, que atiende las solicitudes y en respuesta construye y despacha las páginas. La instalación se realiza ejecutando un script SQL, pero para ofrecer una idea de sus dimensiones, en la versión 5.1 el core de APEX se implementa con más de 400 tablas, y más de 200 paquetes de PL/SQL implementados con más de 425.000 líneas de código. Actualmente Oracle ofrece APEX como una de las alternativas de migración de las aplicaciones obsoletas basadas en Forms y le ha incorporado capacidades de migración semi-automática de Forms a APEX.

Aún después de esta presentación cabe la pregunta de qué es exactamente APEX, y hay varias respuestas según el enfoque con que se mire. Se puede decir que:

- Es un producto, desarrollado y mantenido por Oracle
- Es un *Integrated Development Environment* (IDE) de aplicaciones web, donde el desarrollador solo debe conocer SQL y PL/SQL
- Como entorno de desarrollo es un RAD debido a la velocidad con que se puede prototipar
- Es un LCAP, es decir, un entorno de desarrollo con poco código y a la

vez un entorno de ejecución de aplicaciones, ya que se ejecutan sobre la misma tecnología que el propio IDE

- Es un desarrollo implementado en un RDBMS con un *web listener* que le permite atender invocaciones por HTTP
- Sigue una arquitectura particular, que parece no haber sido analizada en la literatura

Desde el punto de vista de la arquitectura, es posible ver a APEX como el extremo en el cual se mantiene y ejecuta en el RDBMS todo lo que es posible.

Al igual que con Forms, la lógica que hace uso intensivo de los datos se mantiene en el RDBMS como *stored procedures*. A diferencia de Forms, no existe un *Application Server* que entregue los forms y mantenga un proceso de servidor para atenderlos ejecutando alguna lógica, sino que el *Middleware* fue reducido a un componente esencialmente tecnológico que no es responsable de ejecutar ningún tipo de lógica, y la lógica que en Forms se ejecutaba en el *Application Server* en APEX se ejecuta en el RDBMS. Por otro lado, APEX también se diferencia de Forms en que del lado del cliente no hay un cliente grueso, sino un *web browser*, y algunas funcionalidades de interfaz son implementadas con *JavaScript*. En este caso sí podemos decir que esta arquitectura cumple con los requisitos de una aplicación web que establecimos en la Sección 2.1. Resulta interesante comparar las vistas lógicas de Forms y APEX de las figuras 2.8 y 2.9, observando que tenemos los mismos tipos de componentes conceptuales, pero mientras en el caso de Forms los componentes de lógica estaban distribuidos entre en *application server* y el *database server*, en el caso de APEX toda la lógica está ubicada en el *database server*. Vale destacar, sin embargo, que APEX no es una evolución de Forms, sino simplemente una alternativa.

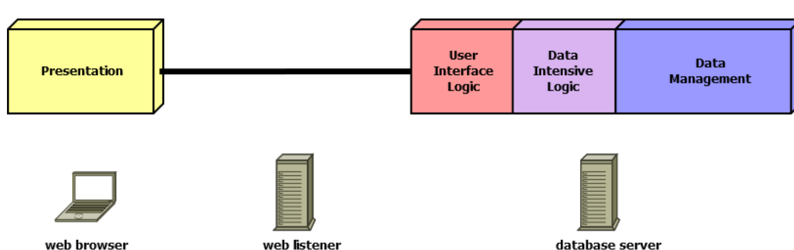


Figura 2.9: Vista lógica de la arquitectura de APEX. Producción propia con base en [51]

Si bien existen trabajos enfocados en arquitecturas *database-centric*, donde

el RDBMS no queda relegado a ofrecer un servicio de persistencia sino que juega un rol central en la arquitectura, no se encontraron trabajos que presenten el extremo donde toda o casi toda la arquitectura quede comprendida en el RDBMS. Por otro lado, existen trabajos donde se hace referencia a APEX, pero no incluyen un análisis detallado de su arquitectura. En la Sección 2.5 se presenta un resumen de los trabajos que hacen referencia a APEX, mientras que en la Sección 2.6 se comentan los trabajos más relevantes sobre las arquitecturas *database-centric*.

2.5. Trabajos relacionados con Oracle APEX

En la literatura hay muy pocas referencias de APEX, y la mayor parte de ellas son evaluaciones de su adaptación al uso.

Krishna et al [55] proponen una metodología y una herramienta para analizar la evolución de las aplicaciones *database-centric* y específicamente el caso de APEX. Los autores no definen el concepto de *database-centric* y presentan a APEX como una herramienta RAD, con ventajas para la utilización de metodologías ágiles pero con desventajas en lo que respecta a la gestión de cambios. La propuesta consiste en comparar automáticamente dos versiones de una aplicación APEX mediante un *diff* inteligente en varios niveles, considerando las jerarquías de elementos de APEX (*application level diff*, *page level diff*, *region diff*, *item diff*, *schema dependency diff*). Desarrollaron una herramienta que es en sí misma una aplicación APEX llamada *Evolving Application Synopsis Tool* (EAST) que permite navegar por una descripción de los cambios, mediante la implementación de medidas de similaridad de regiones y algoritmos para calcular los cambios en las páginas. Como marco conceptual, identifican diferentes jerarquías de Vista y Modelo de una aplicación APEX bajo el patrón *Model–View–Controller* (MVC). La jerarqía de Vista se compone de: *application* → *web pages* → *page regions* → *region items*; mientras que la jerarqía de Modelo se compone de: *application* → *web pages* → *events* → *event processes* → *schema dependencies* (al Controlador lo ven como una parte inherente del Modelo y por esto su jerarquía no es presentada de forma separada). La propuesta resulta interesante tanto por el hecho de poder ver gráfica y fácilmente todos los cambios entre dos versiones de una aplicación, como por la posibilidad de contar con métricas sobre estos cambios. Con esta herramienta es posible contestar preguntas como ¿en qué

versión de una aplicación se comenzó a referenciar una determinada tabla? o ¿cuándo se eliminó un determinado ítem?. Los autores planeaban extender la herramienta para considerar los cambios en el esquema de la base, y adaptarla a otros entornos, como el de una aplicación PHP, aunque advierten que esta tarea sería desafiante por la ausencia de información de metadata de las aplicaciones, tal como sí tiene APEX. Para el propósito de comprender la estructura de una aplicación APEX, este trabajo resulta interesante por la utilización del patrón MVC en el análisis así como por la jerarquía que proponen para la Vista y el Modelo.

Monger et al [56] analizan el uso de APEX en el marco de un curso de introducción a las bases de datos. Lo describen como una herramienta web gratuita para desarrollar aplicaciones web, y destacan el hecho de que se pueden crear fácilmente y mediante asistentes páginas principales, de *login*, formularios y reportes, que pueden después transformarse en una aplicación funcional agregando PL/SQL. Destacan las facilidades de administración y monitoreo de la actividad de los estudiantes, así como el entusiasmo de éstos en el uso de la herramienta, calificando la experiencia como exitosa. Sin embargo, han evidenciado también algunos problemas que experimentaron, por ejemplo, que los workspaces compartidos limitan el desarrollo del esquema y posibilitan el plagio (problemas derivados de la decisión de utilizar workspaces compartidos); y la imposibilidad de utilizar claves compuestas (limitación superada en las últimas versiones). Este es un caso de evaluación positiva de la adaptación al uso, con un énfasis en la utilidad para la enseñanza y práctica de conceptos de bases de datos.

En otro análisis llevado a cabo en el CERN, también de 2009, Zaharieva et al. [57] analizan las ventajas de APEX como herramienta que permite prototipar, construir y poner en producción aplicaciones de forma rápida, para responder a un escenario de requerimientos cambiantes. Describen a APEX como un entorno gráfico de desarrollo web *Fourth-Generation programming Language* (4GL). Mencionan que el CERN ha tenido grandes desafíos en cuanto a las necesidades cambiantes de reporting sobre bases de datos grandes, y por parte de múltiples grupos con necesidades diferentes; y eligieron APEX en 2004 por sus capacidades y debido a que previamente utilizaban *Oracle Web Access* (OWA) y Oracle Database era el RDBMS estándar. Comentan que el entorno dinámico del CERN requería un entorno de programación ágil, de prototipado rápido, y con ciclos rápidos y simples de entrada en producción,

todo lo cual era ofrecido por APEX, con componentes listos para usar y asegurando una interfaz con apariencia profesional. Por otro lado, destacan que APEX ofrecía la estabilidad y confiabilidad requerida para las operaciones del CERN, y que la adopción de APEX fue sencilla debido al dominio previo de SQL y PL/SQL, así como de *Oracle Forms* y de HTML, *Cascading Style Sheets* (CSS), JavaScript y *Asynchronous JavaScript and XML* (AJAX). Del 2004 al 2009 cada aplicación de reporting fue desarrollada utilizando APEX, siendo más de 10 las aplicaciones desarrolladas. Concluyen que APEX tiene muchas ventajas, sobre todo cuando ya se tiene disponible el RDBMS Oracle. Evalúan positivamente el agnosticismo respecto del sistema operativo y el browser del cliente; así como las buenas experiencias en las múltiples migraciones desde la versión 1.6 en 2004 hasta la 3.2 en 2009. Como desventaja, mencionan la ausencia de un sistema integrado de control de versiones (desventaja que se mantiene hasta hoy, y cuestión que supone no pocos desafíos). Consideran por todo esto a APEX como una herramienta madura en 2009. Una presentación del CERN de 2015 permite saber que continuaron utilizando APEX, que llegaron a integrarlo a su Single Sign On y a su nube privada, en el marco de un proyecto interno llamado *Middleware On Demand* (MWOD), como una verdadera solución de *Platform as a Service* (PaaS) donde el desarrollador se convierte en el dueño del servicio [58].

Wang y Kourik [59], en 2012, presentan otro caso de uso de APEX como plataforma para el aprendizaje de tópicos de bases de datos. Sostienen que los cursos sobre bases de datos, sobre todo los avanzados, son dependientes de la tecnología, y que la experimentación con bases de datos reales y SQL aumentan la efectividad del aprendizaje. Para evitar que los estudiantes deban lidiar con problemas de instalación de herramientas, los autores desarrollaron para la universidad de Webster, laboratorios que pudieran ser accedidos a través de Internet. En particular, en los cursos sobre aplicaciones sobre bases de datos, argumentan que necesitaban trabajar con aplicaciones de tipo *data-driven*, con interfaces profesionales y seguridad, para poder construir una experiencia similar a las que se ven en la industria. La mayor parte de las alternativas disponibles de bases de datos con soporte para stored procedures implican instalar clientes o IDEs gruesos, mientras que APEX ofrecía un entorno web que cumplía con todos los requerimientos para los laboratorios. Los estudiantes disponían de un entorno web al que accedían únicamente con el *web browser*, que les permitía no solo ejecutar sentencias y *scripts* (incluyendo definición

de *stored procedures*), sino desarrollar, desplegar y ejecutar aplicaciones web. La utilización de APEX para estos laboratorios tenía ventajas frente a otras tecnologías, como el acceso transparente, la reducción del tiempo dedicado a resolver problemas técnicos de instalación, configuración o conectividad, y con ello, el aumento del tiempo destinado al aprendizaje. Como desventajas mencionan la curva de aprendizaje de la herramienta, los desafíos de instalación y configuración del servidor para los administradores, y la inversión en el desarrollo de los materiales del curso; aunque no contrastan estas desventajas, en términos comparativos, con eventuales alternativas. Concluyen que los laboratorios en APEX han sido efectivos en los cursos relacionados a bases de datos, aunque deben utilizarse en conjunto con otras herramientas para maximizar el aprendizaje.

En un artículo de 2013, Austwick [18] presenta a APEX como una plataforma RAD con una *"arquitectura única"*, que presenta desafíos de seguridad que no están presentes en otras tecnologías. Al contar con un sistema integrado de control de las sesiones y autenticación, ofrece una forma fácil de implementar la seguridad más básica, pero al permitir a los desarrolladores trabajar con SQL dinámico y generar HTML, un sistema desarrollado en APEX podría quedar expuesto a ataques de *SQL injection* y *Cross-Site Scripting* (XSS). La arquitectura de APEX y su acoplamiento en la base de datos hace que sea sobreanalizada en las evaluaciones de riesgos y los tests de penetración. Esto ha llevado a la creación de *plugins* de Nessus, para ayudar a las organizaciones a evaluar la seguridad de sus aplicaciones APEX. Concluye que no hay nada que haga a las aplicaciones APEX inherentemente menos seguras que las implementadas con otras tecnologías, pero hay que tener un inventario de las aplicaciones, comprender los riesgos, educar a los desarrolladores en seguridad y realizar tests de seguridad dentro del ciclo de vida del desarrollo [60].

Si bien las evaluaciones sobre la adaptación al uso de APEX son en general favorables, llama la atención las pocas referencias explícitas a esta tecnología.

2.6. Trabajos con el enfoque database-centric

No hay definiciones aceptadas en la literatura para los términos *database-centric* y *database-driven*, si bien existen trabajos que los mencionan. Para este trabajo asumiremos que el término *database-centric* hace referencia a una

arquitectura de software donde existe un DBMS que juega un rol central para la lógica de las aplicaciones, mientras que *database-driven* hace referencia a una arquitectura de software *database-centric* donde los datos almacenados en la base se utilizan para determinar la lógica y comportamiento de las aplicaciones. Existen otros términos similares para hacer referencia a sitios web que están construidos "sobre una base de datos", como *database-backed web sites* o *RDBMS-backed web sites* [61].

Junto con el rol central del DBMS en la lógica de las aplicaciones, el enfoque *database-centric* típicamente también prescribe una metodología para el proceso de desarrollo de software que comienza por el diseño del modelo de datos y las transacciones en un RDBMS, seguido por el diseño de un flujo de páginas, y terminando por la implementación de las páginas individuales [61, 62].

En un trabajo de 2000, Shafer et al [63] muestran que en una aplicación *database-centric*, es posible que el enfoque clásico basado en la metáfora de los ciclos de *submit/response* pueda ser mejorado implementando alguna metáfora mejorada como la del *continuous querying*, con capacidades de exploración más allá del refinamiento de las condiciones de búsqueda, y presenta la arquitectura de un prototipo con estas capacidades. La idea básica es que en la parte superior de cada reporte se incluyan elementos de interfaz para controlar las restricciones sobre cada columna, idea que de hecho es utilizada en los *Interactive Reports* de APEX, pero la propuesta incluye otras capacidades como la búsqueda por similaridad, donde se pueden seleccionar tuplas de ejemplo y especificar para cada columna si se desea refinar la búsqueda por los mismos valores, valores cercanos (según una función de similaridad), más valores o menos valores. A nivel de diseño, la principal decisión es la de mantener el *resultset* del lado del cliente, y sobre esta caché mantener un estado, representado por los valores de los atributos de control y las tuplas seleccionadas. Como las modificaciones del estado se realizan con el mouse y los tiempos de cambio de estado se dan a intervalos humanos, entienden razonable que ante cada cambio de estado se realice una actualización interactiva sobre el *resultset*. Además, argumentan que como el usuario nunca puede ver más tuplas de las que entran en una página, los resultados se pueden paginar y el *resultset* puede no ser nunca mayor al tamaño de una página, lo que permite que el sistema tenga tiempos de respuesta excelentes independientemente del tamaño del *resultset* virtual de

la consulta. Los datos se mantienen internamente como columnas en lugar de la tradicional estructura basada en filas, lo que permite que las columnas se carguen de forma idependiente y asincrónica. Un problema de este prototipo es que está basado en las funcionalidades de un componente grueso especialmente diseñado, y debería analizarse cuidadosamente la posibilidad e implicancias de cambiar los componentes de cliente grueso por elementos correspondientes en una aplicación web que pueda ser utilizada mediante un *web browser* (e.g. JavaScript, AJAX y datos como *JavaScript Object Notation* (JSON)) y permita actualizaciones parciales.

Haciendo foco en el desempeño, en un trabajo de 2001, Candan et al [64] notan que la mayoría de los *Application Servers* deben marcar las páginas generadas dinámicamente como no cacheables, debido a que no saben si los datos generados de forma dinámica seguirán siendo correctos; y presentan la arquitectura de CachePortal: un sistema que permite el *caching* de contenido dinámico para aplicaciones web *database-driven*. Concluyen que las ideas de este trabajo bien podrían ser adoptadas en cualquier aplicación *database-driven*, aunque la relación costo/beneficio de adoptar un sistema complejo como el descrito solo puede ser favorable cuando el desempeño sea un factor realmente crítico y se den ciertas condiciones sobre la probabilidad de reutilización del contenido cacheado, lo que depende sobre todo del tipo de aplicación.

En un trabajo de 2007, Ploski et al [65] advierten sobre la atención que deben poner, aún las empresas pequeñas, en el control de versiones cuando adoptan nuevas tecnologías para aplicaciones web; si bien el objetivo primario suele ser la utilidad, el aspecto y la rapidez. Más allá de la facilidad de adoptar un repositorio de control de versiones para el código de las aplicaciones y metodología para utilizarlo correctamente, está el hecho de que revertir cambios en una base de datos es inherentemente más complejo que cambiar una versión de código de aplicaciones por una anterior. Como solución, proponen contar con ambientes clonados de producción y agregar en el sistema de control de versiones los archivos SQL con las sentencias para modificar el estado de la base, de forma que puedan ser testeados antes del pasaje a producción. Este agregado exige mucha disciplina por parte de los desarrolladores, y constituye un método propenso a errores. Para superar estas dificultades, desarrollaron herramientas específicas para exportar la estructura de las tablas de la base, para comparar estos datos exportados y para generar los scripts necesarios

para pasar de un estado al otro. Mediante estas herramientas, el estado de la base en formato texto puede ser gestionado por el controlador de versiones como si fuera código de aplicaciones. Advierten que esta metodología tiene riesgos y limitaciones varias, fundamentalmente porque la herramienta que implementa las comparaciones, al no ser trivial, no soporta *refactoring* que incluya modificaciones de la instancia ni soporta todos los posibles cambios de esquema. Concluyen que las herramientas gratuitas disponibles para gestión de la configuración no son suficientes, que las mejores prácticas en gestión de la configuración para aplicaciones web *database-centric* no están disponibles de forma gratuita, y que de la cooperación de las pequeñas y medianas empresas con la academia podría resultar un mutuo beneficio. Finalmente, destacan que la automatización tiene el objetivo de prevenir errores triviales y faciltar las tareas ordinarias, pero todo proceso de automatización debe incorporar excepciones y no sustituye las competencias individuales.

En dos trabajos de 2013, Grust et al [66, 67] proponen la idea de que las funciones PL/SQL también son datos, y pueden ser tratadas como tales. Este paradigma de "funciones como datos" implica tratar a las funciones como datos de cualquier otro tipo, que puedan ser: (1) construidas en tiempo de ejecución, (2) mantenidas y recuperadas de tablas, (3) asignadas a variables y (4) pasadas como parámetros de otras funciones de mayor orden. Demuestran que este tratamiento genera un nuevo lenguaje, de estilo funcional, que complementa la práctica existente pero además permite un nuevo dialecto particularmente conciso, elegante y eficiente para problemas típicamente funcionales como el cálculo de clausuras transitivas. Para implementar este paradigma es necesaria una extensión a los RDBMSs existentes, y los autores realizaron pruebas con PostgreSQL. Sin embargo, es interesante notar que para los puntos 1 y 2 mencionados anteriormente, no es necesaria ninguna extensión, y se podría obtener alguna ventaja de tratar a las funciones como datos que pueden construirse en tiempo de ejecución.

En un trabajo de 2014, Cheung et al [68] presentan Sloth, una propuesta para mejorar el desempeño de aplicaciones que utilizan bases de datos, demorando el cómputo utilizando semánticas *lazy* para eliminar *round-trips* entre un servidor de aplicación y la base de datos. Si bien este trabajo no aplica de forma directa al caso de una arquitectura *RDBMS-only*, ya que no existe el nivel del servidor de aplicaciones, ni existe código pasible de ser precompilado por la herramienta propuesta, se realizan en él algunas

afirmaciones interesantes. Una de ellas es que parte de la latencia de las aplicaciones se debe a que un *page load* típico invierte un tiempo significativo en la ejecución de sentencias y la espera por los *round-trips* de red para completarse, lo que es exacerbado por los *frameworks* de ORM cuyos *mappings* terminan generando sentencias ineficientes. Otra afirmación interesante es que una forma de mejorar el desempeño es la evaluación *lazy*, es decir, demorar la ejecución de las sentencias hasta que sean absolutamente necesarias. Vale destacar que en el caso de APEX o de cualquier aplicación con una arquitectura semejante, no existe un componente de tipo ORM que realice *mappings* de forma automática y pueda generar sentencias ineficientes, sino que las sentencias deben ser escritas por el desarrollador. Esto último no es una garantía de que las sentencias sean eficientes, pero al menos evita que estas puedan ser ignoradas por el desarrollador. En APEX, también ocurre que cuando una página realiza un *submit*, todas las sentencias que este *submit* genera son absolutamente necesarias, es decir, el modelo es de evaluación *lazy* por diseño, ya que en los términos del patrón MVC el desarrollo en APEX se puede ver como *view-driven*. Estas consideraciones hubieran sido difíciles de expresar (y tal vez aún de descubrir) sin la ayuda de la comparación de conceptos entre APEX y la propuesta de Sloth.

Finalmente, en un trabajo muy interesante de 2004, titulado "A Database-Centric Approach to J2EE Application Development", Koppelaars [69, 70] describe el problema que tienen los desarrolladores con conocimientos de PL/SQL para dominar la programación orientada a objetos y la complejidad tecnológica de un framework como Jakarta EE. Menciona que las posibilidades del RDBMS Oracle (que fue sobre el que trabajó, pero podríamos extender su argumento a otros RDBMSs) permiten que toda la lógica de una aplicación sea implementada en la base, y preferir PL/SQL a Java para esta lógica, permite tener una capa muy fina de Java para la interfaz, reduciendo los costos y los riesgos de los proyectos. El autor también realiza una clasificación interesante de las ocho alternativas de los sistemas de tres capas (*client-tier*, *middle-tier*, *data-tier*) según cada una de estas capas sea fina (*thin*) o gruesa (*fat*), notando además que el caso *thin/thin/thin* no tiene interés real. Al caso *thin/fat/thin* se le podría llamar *"Thick Middleware"* mientras que al caso *thin/thin/fat* se le podría llamar *"Thick Database"*. La Figura 2.10 presenta estos dos casos lado a lado, dentro de un esquema cliente-servidor.

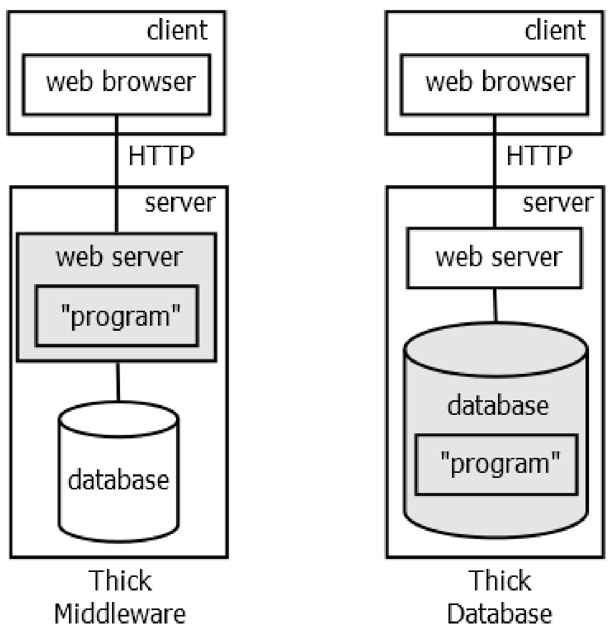


Figura 2.10: *Thick Middleware* vs *Thick Database*

Según esta clasificación, además, discute dónde estarían ubicados los componentes de *Model*, *View* y *Controller* del patrón MVC en cada uno de los casos. Otra clasificación interesante es la del código, en: *User Interface (UI) Code*, *Business Logic (BL) Code* y *Data Logic (DL) Code*. La propuesta *database-centric* se centra en el diseño de las aplicaciones como una red de páginas, cada una de las cuales de alguna forma mapea el concepto de *Enterprise JavaBeans* (EJB) y se implementan con una API que mapea *UI Code* a *BL Code* utilizando *policy functions*, *instead-of triggers* y *dynamic views* (características disponibles en Oracle Database). La arquitectura general de esta propuesta es sorprendentemente parecida a la de APEX, aunque también tiene varias diferencias y parece ser un desarrollo independiente. El autor concluye que este enfoque permitió adoptar Java como lenguaje para una interfaz web, sin hacer las cosas más difíciles de lo que podrían haber sido con un enfoque tradicional, así como bajar el *Total Cost of Ownership* (TCO) de las aplicaciones, especialmente bajando los costos del mantenimiento [71]. Por supuesto, todo esto atentando contra la portabilidad y la promesa *"write once, run anywhere"*, pero en definitiva permitiendo la portabilidad a otro lenguaje de propósito general que no sea Java. Termina preguntándose cuál de los dos lenguajes habrá venido para quedarse más tiempo, si Java o PL/SQL, y los años que han pasado desde 2004 hasta ahora aún no permiten responder a esa

pregunta de forma concluyente.

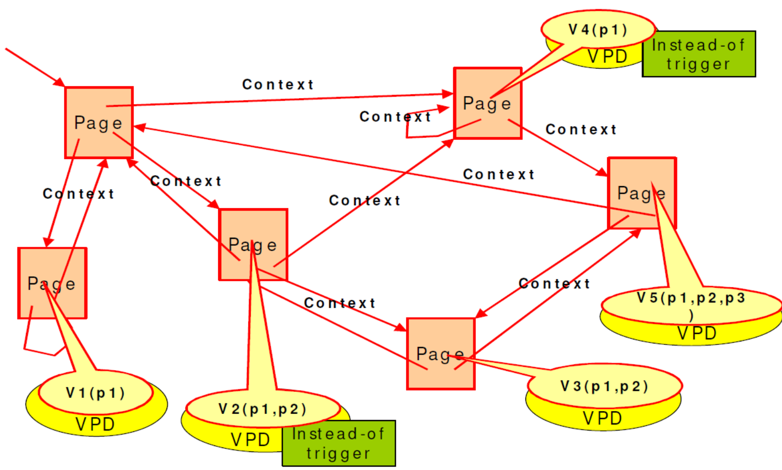


Figura 2.11: Arquitectura de la propuesta *database-centric* de Koppelaars [69]

Es interesante pensar en una extensión del análisis de este trabajo, según la cual una arquitectura *RDBMS-only* podría ser considerada del tipo *thin/zero/fat*, haciendo desaparecer la *middle-tier*, y teniendo tanto el *Model*, como el *View* y el *Controller*, todos en el RDBMS, ya que los entornos de ejecución para una arquitectura *RDBMS-only* serían SQL y un lenguaje procedural dentro del RDBMS, y JavaScript del lado del cliente, dentro del *web browser*. En el caso de la arquitectura *RDBMS-only*, tanto el *BL Code* como el *DL Code* se ejecutarían siempre en el RDBMS, así como la mayor parte del *UI Code*, dejando solo parte del *UI Code* que podría ser JavaScript, mantenido en el RDBMS pero ejecutado en el *web browser* del cliente. Por motivos de usabilidad y desempeño, algunas restricciones del *DL Code* (e.g. que una fecha de inicio sea menor que una fecha de fin), además de estar en el RDBMS por seguridad, podrían estar replicados en código JavaScript que se ejecute del lado del cliente. Más allá de estas adaptaciones, lo interesante de este trabajo es que parece haber llegado de manera independiente a la conveniencia de un desarrollo *database-centric*, donde la lógica se mantiene en *stored procedures* en el RDBMS y el diseño de la aplicación se realiza en base al concepto de páginas que están fuertemente relacionadas a datos almacenados en tablas en el RDBMS.

Para esta tesis, la propuesta de Koppelaars podría considerarse como un faro, y la evolución de sus ideas pueden seguirse en su blog titulado *The Helsinki Declaration (IT - version)*, donde afirma que el desarrollo

de aplicaciones web se ha vuelto extremadamente complejo, y que las características dentro de los RDBMSs (especialmente los DBPLs) permiten construir aplicaciones web *database-centric* con la lógica en el RDBMS, lo que redunda en aplicaciones más fáciles de mantener y con mejor desempeño [70].

En el contexto de esta declaración, Koppelaars realiza cuatro observaciones relevantes, aunque no publicadas en ningún artículo. Estas observaciones son hipótesis, parecen basadas en su experiencia personal y subjetiva, y no parece que se haya realizado ningún estudio basado en datos para intentar refutarlas, lo que podría explicar que se mantengan en el contexto de un blog. Sin embargo, parecen lo suficientemente relevantes para el tema de este análisis como para reproducirlas aquí, con la advertencia de que son una opinión. La primera es que si bien la oferta de características implementadas dentro de los RDBMSs han crecido significativamente, la demanda de estas características en general acompañó la oferta hasta el año 2000, luego de lo cual se redujo rápidamente al mínimo, mientras crecían las funcionalidades implementadas por fuera del RDBMS. Para ejemplificar esta idea, Koppelaars presenta un gráfico como el que se ha reproducido en la Figura 2.12.

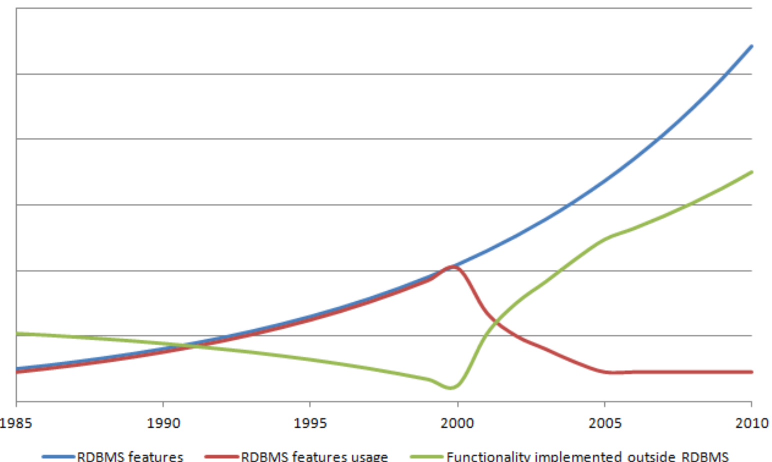


Figura 2.12: Utilización de funcionalidades dentro y fuera del RDBMS. Producción propia con base en [70]

Debe notarse que la medida de las funcionalidades está expresada en una escala desconocida, y el espíritu del gráfico no parece ser reflejar valores reales sino comunicar su idea de la tendencia. El hallazgo que parece querer expresar es que las nuevas funcionalidades de los RDBMSs se fueron utilizando casi al ritmo en que estuvieron disponibles aproximadamente hasta el año

2000, mientras que el número de funcionalidades implementadas por fuera del RDBMS fueron bajando, presumiblemente por sustitución; pero a partir del año 2000, se disparó el número de funcionalidades implementadas por fuera del RDBMS, con el consiguiente decremento en el número de funcionalidades del RDBMS utilizadas, estabilizándose pronto en un mínimo la utilización de las funcionalidades del RDBMS. También es parte de la observación, que esto no detuvo la oferta de nuevas funcionalidades por parte de los RDBMSs. El autor no ofrece una teoría que explique por qué se ha dado el desplazamiento en el uso de funcionalidades del RDBMS a otras tecnologías.

La segunda observación presenta la ubicación general de las aplicaciones según su etapa histórica, en cuatro cuadrantes, según sea su interfaz de caracteres o gráfica, y su protocolo orientado o no a conexión.

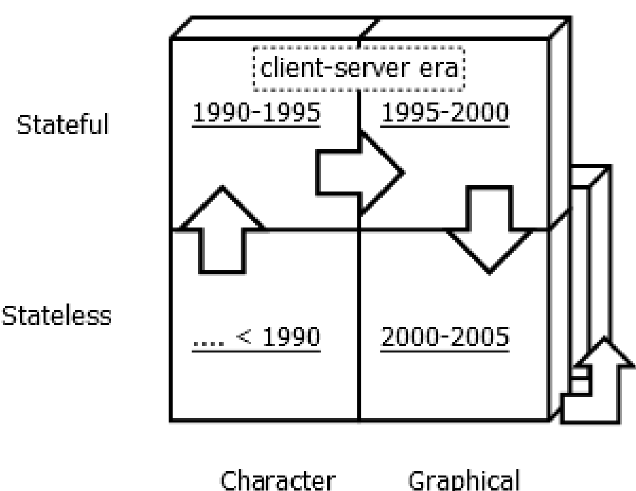


Figura 2.13: Etapas en la arquitectura de software [70]

Antes de 1990 las interfaces eran de caracteres y se operaban programas de servidor desde terminales, a partir de 1990 se entra en la era de las arquitecturas cliente-servidor pudiendo mantener una conexión TCP/IP entre el cliente y el servidor, a partir de 1995 se comienza a generalizar el uso de interfaces gráficas, y a partir de 2000 se generaliza la utilización de web browsers lo que reintroduce la utilización de un protocolo no orientado a conexión como HTTP. El agregado de un protocolo no orientado a conexión entre el cliente y "el servidor" (entendido como el conjunto de servidores y tecnologías que soportan una aplicación) implica que debe agregarse la tarea de mantener el estado y delegársela a algún componente del lado del servidor. En este punto, los RDBMSs pudieron haber sido un candidato para esta tarea, si bien en

general no fue lo que ocurrió, comenzando la popularización de tecnologías de *application servers*. Después de 2005 se continuó en ese cuadrante, simplemente mejorando la interfaz y el desempeño de las aplicaciones web con tecnologías de RIA, de ejecución dentro del browser como JavaScript y de actualización parcial de páginas como AJAX. Se reproduce este cuadro en la Figura 2.13.

La tercera observación es que ha habido una explosión de tecnologías de aplicación fuera de los RDBMSs, para lo cual acuñó el concepto *Yet Another Front End Technology* (YAFET). Por supuesto, podrá objetarse que estas tecnologías ofrecen mucho más que interfaz, pero manteniendo el alcance en las aplicaciones *database-centric*, se pueden ver como tecnologías que se ponen "al frente" del RDBMS (desde el punto de vista del usuario). Koppelaars postula que las aplicaciones actualmente se ven muy diferentes a las de hace dos décadas, pero que los requerimientos de los clientes son esencialmente los mismos, y esta *"demanda incambiada"* no justifica la explosión tecnológica. Podríamos cuestionar que sea razonable haber pasado de *"requerimientos incambiados"* a *"demanda incambiada"*, ya que la hipótesis que parece postular Koppelaars (i.e. que esta demanda no ha sido razonable bajo algún criterio de razonabilidad) no elimina el hecho de que tal demanda haya existido.

La cuarta y última observación se refiere al tiempo de inversión en conocimiento que necesita un desarrollador para producir una aplicación web, y postula que con la utilización de tecnologías como Jakarta EE, la relación entre inversión en conocimiento y producto realizado se ha vuelto cada vez más alta. A su vez, menciona que esto explica que los actuales equipos de desarrollo estén formados por tantos especialistas diferentes, imposibilita que una única persona pueda dominar todas las tecnologías necesarias y exige la intervención de un arquitecto para mediar entre todos los perfiles técnicos. Esta observación se refiere específicamente a la comparación entre tecnologías de desarrollo como Java EE frente a las tecnologías de desarrollo de *Oracle Forms* centradas en la base de datos, y aunque podría tal vez generalizarse para no mencionar tecnologías concretas, podríamos cuestionarnos sobre la validez del juicio de valor negativo sobre la cantidad de especialistas técnicos (explícito en la "observación") en función de los beneficios de la divisón del trabajo, o al menos cuestionar la validez de la crítica en todos los escenarios.

Finalmente, Koppelaars insiste en que algunos requerimientos de los usuarios finales no han cambiado, y aún necesitan realizar tareas como encontrar un cliente, ver los detalles del cliente, ingresar un pedido del cliente,

o emitirle una factura. A este tipo de aplicaciones les llama *Window-on-Data (WoD) Applications*: aplicaciones que proveen "ventanas" sobre los datos. Una ventana en una *WoD Application* permite al usuario consultar los datos relevantes y realizar transacciones sobre los datos, exactamente el tipo de tareas para las que nacieron los RDBMSs. Postula que las aplicaciones web *database-centric* que se construyen actualmente, de formas más complejas cada cinco años, todavía son este tipo de aplicaciones, y se beneficiarían de utilizar ventanas sobre los datos en un RDBMS en lugar de utilizar Programación Orientada a Objetos, lo que ve como un retroceso en cuanto vuelve a mantener juntos los datos y el comportamiento.

El descubrimiento de *The Helsinki Declaration (IT - version)*, fue tardío en el desarrollo de este trabajo, y tiene muchas coincidencias con lo presentado en la Sección 2.2, aunque Koppelaars no relaciona la evolución de la tendencia general al hecho de que la Programación Orientada a Objetos se popularizara antes que la web.

Bryn Llewellyn, el gerente de producto de PL/SQL en Oracle ha escrito sobre la comparación entre dos paradigmas extremos: el de *NoPlsql* por un lado y el de *Thick Database* (ThickDB) por otro. El paradigma de *NoPlsql* supone utilizar la base de datos únicamente como una capa de persistencia e implica no tener en absoluto PL/SQL en la base de datos Oracle, y la ausencia de cualquier característica específica de Oracle, por lo cual la única interacción con el RDBMS es a través de sentencias `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `COMMIT` y `ROLLBACK` usualmente generadas por un *framework*. En contraste, ThickDB implica que cada llamada desde el cliente solo invoca a un bloque de PL/SQL, mientras que todas las sentencias `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `COMMIT` y `ROLLBACK` se realizan desde PL/SQL. Esto último significa también que todas las sentencias de SQL y el diseño de la base son producto de un trabajo humano, y no de una pieza de software. Describe asimismo que se desarrollaron experimentos para comparar ambos enfoques, y que obtuvieron resultados que demuestran que el enfoque de ThickDB redunda en mucho menos consumo de recursos en el RDBMS que el enfoque de *NoPlsql*. Bryn Llewellyn y Took Koppelaars han publicado estos trabajos en el marco de los grupos de usuarios de Oracle, y conferencias relacionadas a Oracle como el Kscope [72, 73, 74].

En estas charlas, sostienen que el paradigma *NoPlsql* comienza con un diseño orientado a objetos en una *middle-tier*, lo que deriva en el tratamiento

de la base de datos únicamente como un conjunto de tablas cuya interfaz son las sentencias básicas de SQL; mientras que el paradigma ThickDB comienza con el diseño del modelo relacional, lo que deriva en que los clientes solo ejecuten llamadas a stored procedures en PL/SQL mientras que todas las sentencias SQL se realizan desde código PL/SQL. Estos dos paradigmas son mutuamente incompatibles. Además, declaran que por un lado el enfoque de *NoPlsql* puede derivar en problemas de correctitud, seguridad, desempeño y mantenimiento, ya que los desarrolladores podrían enfocarse en las tecnologías externas al RDBMS al extremo de utilizarlo sin conocerlo lo suficiente; y por otro, defienden la superioridad del enfoque ThickDB, formalizando una clasificación del código en capas que convierte al RDBMS en un proveedor de servicios centralizado y confiable, a la vez que aseguran destruir el mito de que mover la lógica de negocios de la capa de datos a la capa de aplicación mejora las posibilidades de escalar [74].

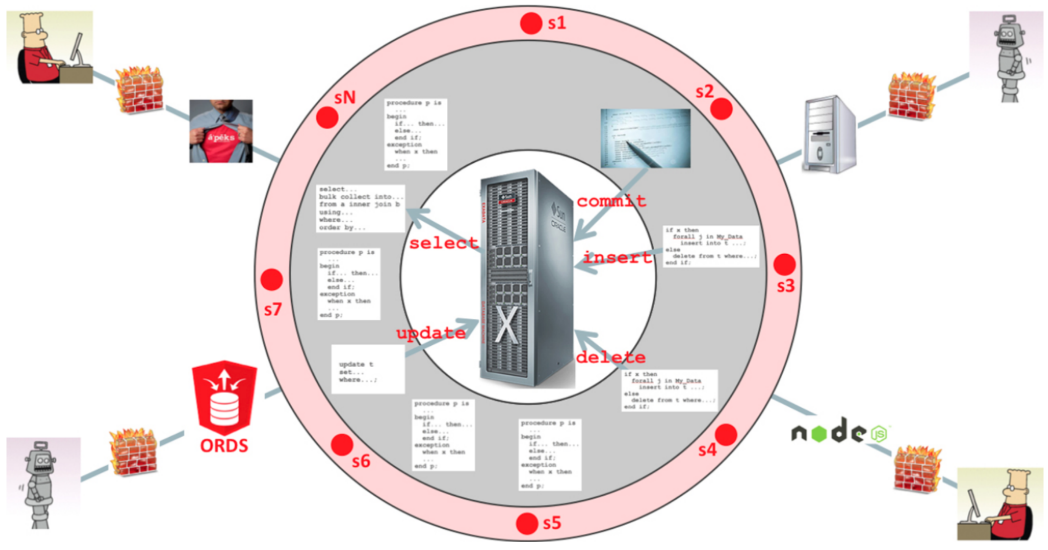


Figura 2.14: El enfoque ThickDB [73]

En la Figura 2.14, el círculo interior representa el esquema de datos, el anillo que está alrededor representa el esquema de código donde se mantiene la lógica de negocios y el anillo exterior representa un esquema de acceso, donde los puntos s1, s2, ..., sN representan sinónimos que denotan el subconjunto de unidades del esquema de código que definen la interfaz expuesta al exterior. Las únicas llamadas permitidas invocan los sinónimos del esquema de acceso. Este enfoque, frente a utilizar la base de datos solo para persistencia o con lógica pero sin una separación en capas o anillos, tiene como beneficios que se facilita

seguir una metodología de diseño por contrato y automatizar pruebas tanto funcionales como de desempeño, así como desarrollar stubs para el desarrollo concurrente de la interfaz y la lógica de negocio.

En un trabajo de 2010, Marius Muji sostiene que la tendencia actual en el desarrollo de aplicaciones está determinada por una presión significativa proveniente de la comunidad de programación, que promueve un enfoque orientado a objetos para toda la arquitectura del sistema de información. En consecuencia, sostiene que las estructuras de datos y las reglas de negocio suelen definirse en un enfoque orientado a funciones, perdiendo su vínculo directo con el modelo de datos, con consecuencias negativas en términos de la flexibilidad del sistema. Concluye que el campo de la ingeniería del software estaba dominado (en 2010) por la nueva tendencia introducida por la arquitectura del OMG *Model Driven* con un fuerte sesgo orientado a objetos, y que existe una clara necesidad de un enfoque orientado a los datos en la ingeniería de aplicaciones [75, 76].

Vale mencionar que algunos de los artículos y libros con el enfoque *database-centric*, como los de Philip Greenspun y Toon Koppelaars están fuertemente orientados a una comunidad de práctica, y casi no han sido citados en trabajos académicos.

Lo característico de las observaciones de Koppelaars y de la propuesta de Llewellyn de sacar más provecho del RDBMS, es que de alguna manera se parecen a las palabras de San Borondón, advirtiendo que aquello sobre lo que algunos estaban parados y creían inerte (el RDBMS) no es una isla sino un animal, e incluso uno de los mayores.

En el siguiente capítulo se analiza Oracle APEX, como caso especial de arquitectura centrada en el RDBMS.

Capítulo 3

Análisis de Oracle APEX

> Yo basaré, pues, mi causa sobre
> mí; soy, como Dios, la negación
> de todo lo demás, soy para mi
> todo, soy el único.
> [...]
> Nada está, para mí, por encima
> de mí.
>
> *El Único y su propiedad*
> *Max Stirner*

En este capítulo se presenta un análisis detallado de la arquitectura de APEX. En la Sección 3.1 se presenta la metodología seguida para el análisis. Las secciones 3.2, 3.3, 3.4, 3.5 y 3.7 presentan respectivamente las vistas lógica, física, de procesos, de desarrollo y de casos de uso de APEX siguiendo el modelo de 4+1 vistas [34]. La sección 3.6 presenta algunas consideraciones especiales sobre una vista de desarrollo para el caso del IDE.

3.1. Metodología

Analizar y describir la arquitectura de un producto complejo es una tarea desafiante de por sí, sobre todo cuando no participamos en el proceso de diseño de este producto. Esto es así, en parte, porque describir una arquitectura supone contestar cómo fue diseñado el producto, teniendo como insumos el producto en sí mismo y la documentación que haya hecho pública el fabricante. Cuando esta documentación no es suficiente, debemos recurrir a alguna técnica

de ingeniería reversa. Si el producto es de código abierto tenemos la ventaja de poder analizar todo el código, de lo contrario podemos vernos impedidos, en mayor o menor medida, de esta posibilidad [77].

APEX, al ser un producto contenido en un RDBMS, tiene visible todo su modelo de información en tablas, y una parte del código PL/SQL que compone la solución también está visible a través del catálogo. Sin embargo, otra parte del código PL/SQL está *wrapped*, esto es, codificado de una forma que hace imposible su lectura directa. Si bien Oracle pretende ocultar el código *wrapped*, técnicamente es posible realizar un *unwrap*. Cabe señalar que Oracle reconoce que esta no es una forma segura de ocultar información sensible, y no resulta fácil saber con certeza si hacer *unwrap* del código de APEX contraviene los términos de licenciamiento que se aceptan al descargar el producto [78, 79, 80].

Lo que sigue es una reseña de la metodología empleada para el análisis de APEX.

Las formas de obtener información relevante para describir la arquitectura fueron:

1. La documentación oficial del producto, así como información obtenida de artículos técnicos y foros de Oracle [81, 82, 83, 84, 85].
2. La utilización del producto, y en particular de la característica de *debug* y las vistas del diccionario de APEX (*APEX Dictionary Views*) [86].
3. El análisis del código PL/SQL que no está *wrapped*
4. La utilización de un *sniffer* para registrar y decodificar todo el tráfico HTTP
5. La utilización de *triggers* de *Data Manipulation Language* (DML) en las tablas de APEX para registrar los cambios en el modelo de información
6. El agregado de *logs* en el código PL/SQL que no está *wrapped* para registrar los flujos de código

3.2. Vista lógica

La vista lógica describe el modelo de diseño de objetos, cuando se utilizan metodologías o lenguajes orientados a objetos, pero puede utilizarse otra forma de vista lógica, como un MER cuando la aplicación es *"data-driven"*. Para este caso se utilizará un MER y las entidades se basarán fundamentalmente en las vistas del diccionario de datos de APEX.

Como se puede observar en la Figura 3.1, existe una jerarquía de conceptos fundamentales organizados según una relación de composición. Una entidad de *Workspace* se compone de varias entidades *Application*, y permite mantener información compartida entre varias aplicaciones como el mapping con esquemas de la base de datos, o los usuarios finales y desarrolladores. Una entidad de *Application* se compone de varias entidades *Page*, y permite mantener información de la aplicación como templates que definen la interfaz gráfica, listas para menúes y barras de navegación, y sistemas[1] de autenticación y autorización. Una entidad de *Page* se compone de varias entidades *Region*, y permite mantener información de la página como procesos que permiten realizar validaciones, modificar el comportamiento de la página, realizar redirecciones de HTTP o ejecutar código en la base de datos. Una entidad de *Region* se compone de varias entidades *Button* y varias entidades *Item*, que modelan los componentes de la interfaz gráfica. Vale mencionar que el modelo que se presenta está muy lejos de ser completo, y apenas representa los conceptos fundamentales del modelo completo de APEX.

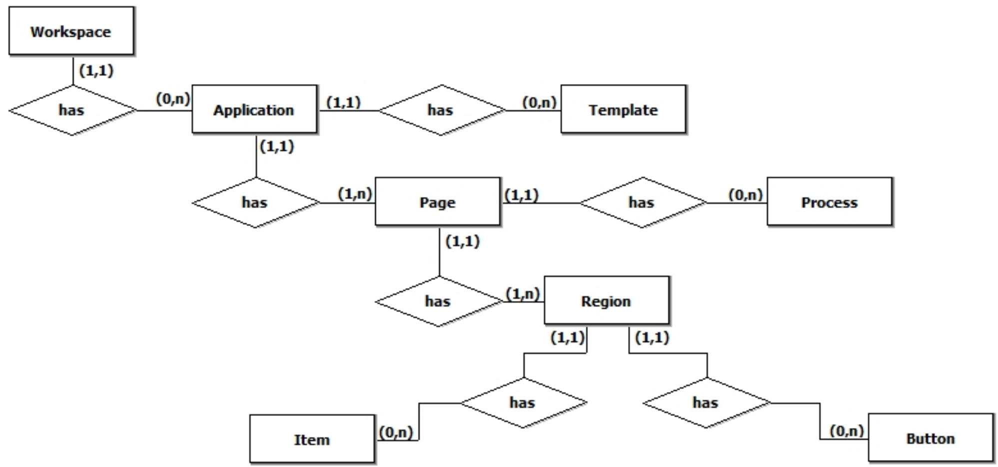


Figura 3.1: Vista lógica de APEX (simplificada)

La entidad *Application* tiene asociadas varias entidades *Template*, que modelan plantillas que regulan la apariencia de la interfaz gráfica. Vale mencionar que estas plantillas mantenidas a nivel de la aplicación son copias de plantillas estándar, para que sea posible la modificación de las plantillas (y con

[1]Se tradujo *scheme* como *sistema*, para evitar confusiones con *schema* que fue traducido como *esquema*. La traducción de *schema* como *esquema* es tradicional, al punto que "esquema de autenticación" podía inducir a pensar que se trataba de un esquema de base de datos.

esto, la personalización de la apariencia) limitada al alcance de la aplicación.

A continuación se discuten los atributos más relevantes de estas entidades fundamentales de la jerarquía. Los nombres de los atributos se tomaron de los nombres de las columnas de las vistas del diccionario de APEX.

En la entidad *Workspace* solo destacaremos dos atributos determinantes: *workspace_id* y *workspace* (el nombre del *workspace*). Como esta entidad es fundamentalmente una agrupación lógica de aplicaciones, no entraremos en más detalles sobre ella, aunque la mantendremos para no modificar la jerarquía básica de conceptos de APEX.

La entidad *Application* modela el concepto de aplicación. Tiene atributos determinantes *application_id* y *application_name*, y se destacan atributos como *theme_number* (una referencia al tema de la aplicación), *owner* (una referencia al usuario dueño de la aplicación), *logo*, *authentication_scheme* (una referencia a un sistema de autenticación), *authorization_scheme* (una referencia a un sistema de autorización) y atributos con URLs generales de la aplicación como *home_link*, *login_url* y *logout_url*.

La entidad *Page* modela las páginas dentro de una aplicación. Tiene atributos determinantes *page_id* y *page_name*, y se destacan además atributos como *page_title* (título visible de la página), *page_alias* (lo que permite referenciarla en una URL), *page_template* (una referencia al template de la página), *page_requires_authentication* (lo que permite discriminar entre páginas públicas y privadas), y otros atributos de contenido de la página que se describen solos como *header_text*, *body_header*, *footer_text*, *help_text*, *page_html_header* y *page_html_onload*.

La entidad *Region* modela regiones dentro de una página. Tiene un atributo determinante *region_name*, y se destacan además atributos como *template* (una referencia al template de la región), *display_sequence* (un ordinal que permite determinar el orden de las regiones dentro de la página), *icon_css_classes* (una referencia a hojas de estilos en cascada con clases predefinidas de íconos) y *display_position_code* (una referencia a posiciones nominadas dentro de una página).

La entidad *Button* modela los botones de la interfaz gráfica. Tiene un atributo determinante *button_name* y se destacan además atributos como *button_sequence* (que permite determinar el orden de los botones en la interfaz), *image_name* (una imagen opcional para reemplazar el componente de interfaz estándar del botón), *label* (una etiqueta visible para el componente de interfaz

estándar del botón) y *button_action_code* (el código que se ejecutará al presionar el botón).

La entidad *Item* modela los elementos básicos de entrada y salida de información, a partir de los cuales se pueden construir formularios de entrada y salida de información, así como reportes. Tiene un atributo determinante *item_name* y se destacan además atributos como *item_data_type* (el tipo de datos del ítem), *display_as_code* (el tipo de componente de interfaz: *text*, *textarea*, *checkbox*, *hidden*, *popup*, *date picker*, etc), *display_sequence* (un ordinal que determina el orden de los ítems) y otros atributos que se describen solos como *is_required*, *item_element_width*, *item_element_max_length*, *label* y *placeholder*.

La Figura 3.2 presenta los atributos recién discutidos, y no es ocioso repetir en este punto que el modelo presentado es una simplificación del modelo original para hacerlo inteligible en una extensión limitada. Sin esta advertencia, podría suponerse que el modelo de APEX es simple. No lo es, como se menciona en la Sección 2.4, el modelo físico de APEX incluye más de 400 tablas con más de 18.000 columnas en total.

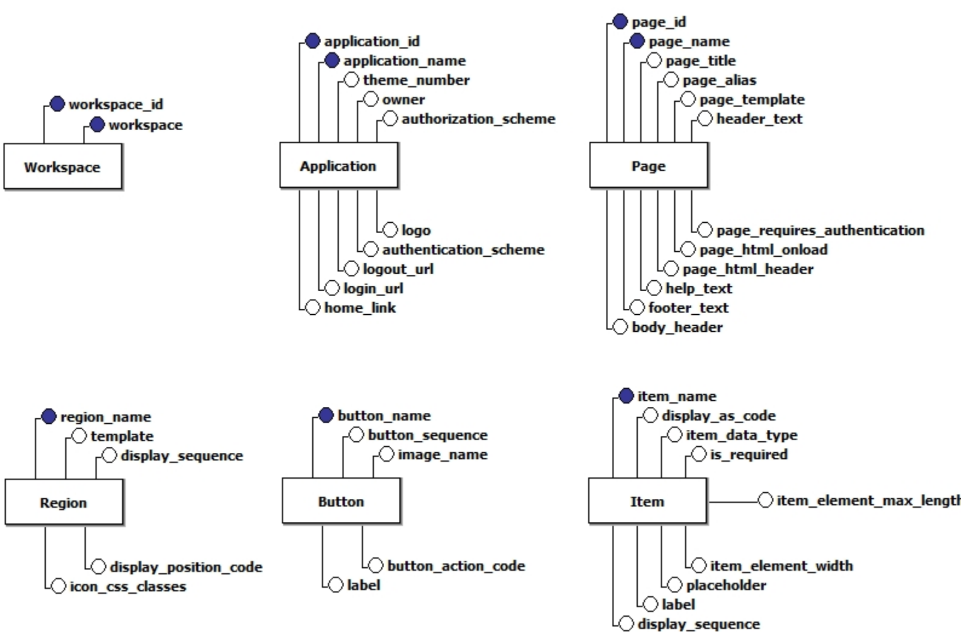


Figura 3.2: Principales atributos de las entidades de la jerarquía de APEX

3.3. Vista física

La arquitectura física de APEX puede concebirse como de dos o tres niveles, siendo el cliente un *web browser*, mientras que el RDBMS actúa como servidor. Puede distinguirse, aparte del RDBMS el *web listener*, un componente fundamentalmente tecnológico que traduce los *HTTP Requests* a PL/SQL, ejecuta PL/SQL en el RDBMS y retorna un *HTTP Response* al cliente. La Figura 3.3 muestra la vista física de la arquitectura de APEX cuando se instala con un Apache HTTP Server y un módulo de Apache llamado *mod_-plsql*, pero la misma vista aplica, por ejemplo, cuando se instala con *Oracle REST Data Services* (ORDS), un *Java Web Archive* (WAR) que se puede ejecutar en modo standalone o desplegado en un *Java Application Server*. Incluso existe la alternativa de que el *web listener* esté implementado en el propio RDBMS Oracle, característica llamada *Embedded PL/SQL Gateway* (EPG) y que extiende el RDBMS para que sea capaz de atender directamente protocolo HTTP por un puerto específico.

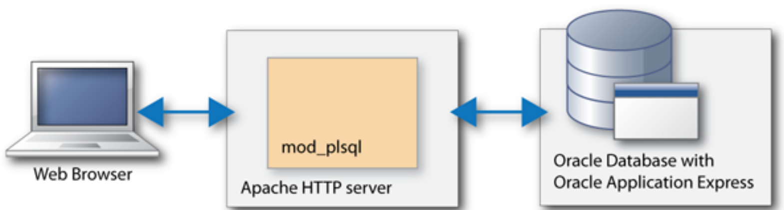


Figura 3.3: Vista física de la arquitectura de APEX con Apache y *mod_plsql*

Es interesante notar que aún en el caso en que el *web listener* esté fuera del RDBMS como en el caso del *mod_plsql* de la Figura 3.3, éste no mantiene estado, y todo el estado de la aplicación se mantiene en el RDBMS. Este hecho de tener un único componente relevante para el estado del sistema de información, tiene implicancias que determinan una relativa simplicidad de enfoque para atender requerimientos comunes. Por mencionar algunos:

- Una estrategia de *Disaster Recovery* (DR) se enfocará en poder restaurar la base de datos
- Una estrategia de *High Availability* (HA) se enfocará en proveer alta disponibilidad a la base de datos
- Una clonación de ambiente se enfocará en clonar la base de datos

- Un análisis de problemas de desempeño se realizará analizando el desempeño en la base de datos

Prescindiendo de los casos típicos de instalación, donde debe instalarse un *web listener*, casi cualquier tarea que deba realizarse en APEX se hará con vistas a la base de datos. De alguna manera, la suficiencia del RDBMS en la vista física de la arquitectura de APEX, puede asimilarse al epígrafe de esta sección aplicado al RDBMS: *"No admito nada por encima de mí"*.

En la documentación oficial de Oracle, cada vez que se refiere a la "simplicidad" de la arquitectura, como en *"Oracle APEX uses a simple architecture that provides zero latency data access, top performance, and scalability [...]"*, por arquitectura parece entenderse la vista física de la misma. [87].

El *web listener* tiene una forma de determinar a qué base de datos debe conectarse en función de un indicador en la URL de los Requests. En el caso del *mod_plsql*, existe un archivo denominado *Database Access Descriptor* (DAD) que permite hacer esta determinación. Es posible entonces que un único *web listener* permita conectarse a entornos diferentes de APEX en bases de datos diferentes, así como también es posible que un mismo entorno de APEX sea accedido de forma concurrente a través de más de un *web listener*. La Figura 3.4 permite visualizar cómo se determina la base a la cual conectarse por medio de un indicador en la URL de los Requests y el DAD.

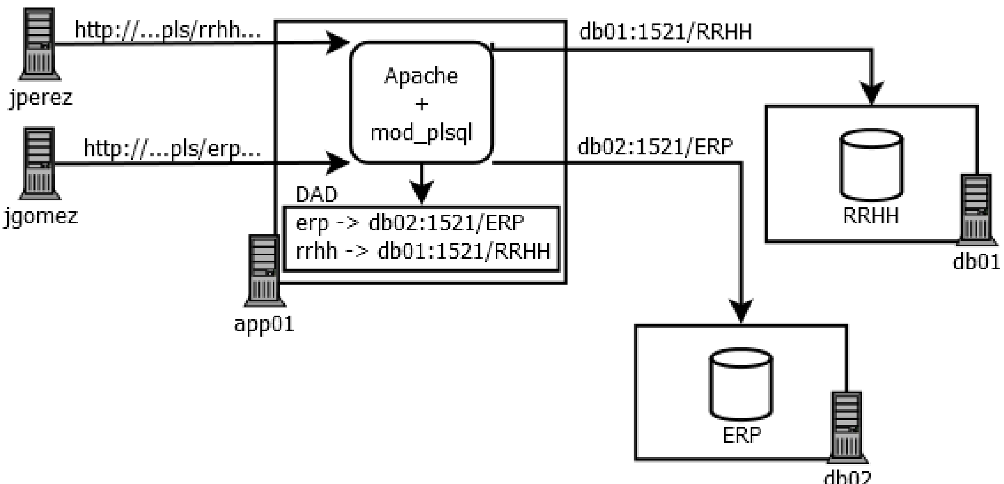


Figura 3.4: Determinación de la base por medio del DAD

En el primer request a una base, el *mod_plsql* crea y luego mantiene un pool

de conexiones con esa base, para que los sucesivos requests no deban pagar el costo de la creación de la conexión.

3.4. Vista de procesos

La vista de procesos describe los aspectos del diseño de la concurrencia y sincronización. Estos aspectos están implementados en el código, por lo que en esta sección deberemos comenzar a analizar la tecnología como una caja blanca. Algo a tener en cuenta para cualquier investigación de caja blanca de APEX, especialmente si involucra un análisis de las tablas y código PL/SQL, es que algunos de los nombres de los objetos de base de datos fundamentales han cambiado con el tiempo, y por este motivo en las tablas de la base se observa una mezcla de nombres antiguos (cuando el producto se llamaba Oracle Flows) y nuevos. La Tabla 3.1 muestra la correspondencia de nombres antiguos y nuevos.

Nombre antiguo	Nombre nuevo
Company	Workspace
Flow	Application
Step	Page
Plug	Region
Instance	Session

Tabla 3.1: Cambios en la terminología de APEX

De esta relación de conceptos también se podrían inferir algunas ideas sobre la concepción original y la evolución APEX. Un *Workspace*, que antes era llamado *Company*, probablemente fue pensado para que varias empresas, o varios departamentos de una empresa grande, pudieran utilizar la misma infraestructura de base para desarrollar sus aplicaciones de forma totalmente independiente. El cambio de *Flow* a *Application*, sugiere que se puede ver la navegación por una aplicación como un flujo de páginas, y el cambio de *Step* a *Page*, que cada página se puede ver como un paso dentro del flujo. Este flujo no tiene por qué ser lineal, y más bien se puede concebir como un grafo dirigido, donde los nodos son las páginas, y las aristas son las transiciones válidas entre páginas. De la misma forma, el cambio de *Plug* a *Region* sugiere que se puede visualizar una región como un componente que se “enchufa” en una página, o dicho de otra forma, que una página es una composición de

regiones. Finalmente, el cambio de *Instance* a *Session*, sugiere que se puede ver una sesión como una instancia de ejecución de un flujo.

Veremos ahora cómo se realiza la dinámica de un request sobre la vista física. Consideraremos el caso del *web listener* implementado con Apache y *mod_plsql*, y para simplificar comenzaremos con el caso del HTTP GET. Un request se produce cuando un usuario solicita, a través de un browser, una URL como la siguiente:

`http://host:7778/pls/apex51/f?p=101:1:220883407765693447`

En este ejemplo:

- host:7778 es la pareja host:puerto donde atiende el *web server* Apache
- pls es el indicador que le dice al *web server* Apache que debe delegar el *HTTP Request* al *mod_plsql*
- apex51 es una entrada del DAD de donde el *mod_plsql* obtiene la información de conexión a la base
- f?p= es el prefijo estándar de APEX, indica que se debe ejecutar el stored procedure llamado f, y que sigue un parámetro llamado `p`. Este parámetro `p` es un *string* que contiene una lista posicional de valores separados por el caracter ":"
- 101 es el identificador de la aplicación APEX
- 1 es la página dentro de la aplicación que debe ser obtenida
- 220883407765693447 es el identificador de la sesión

En la Figura 3.5 podemos ver la dinámica que se genera para procesar un *HTTP Request* con una URL como la anterior.

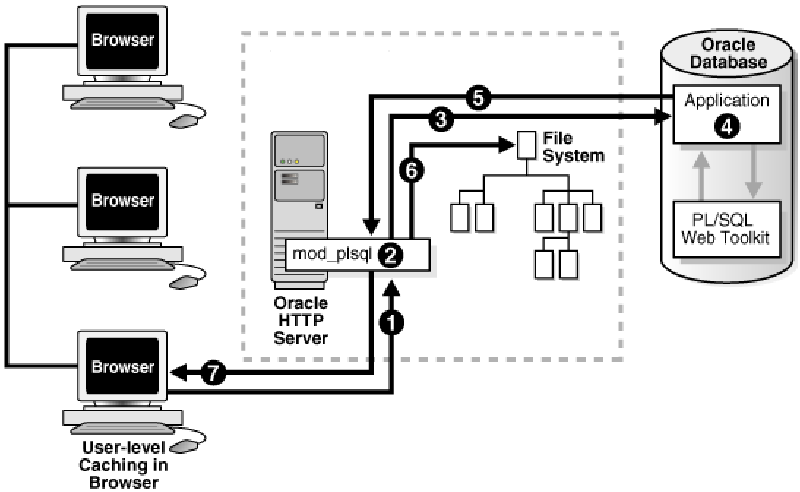
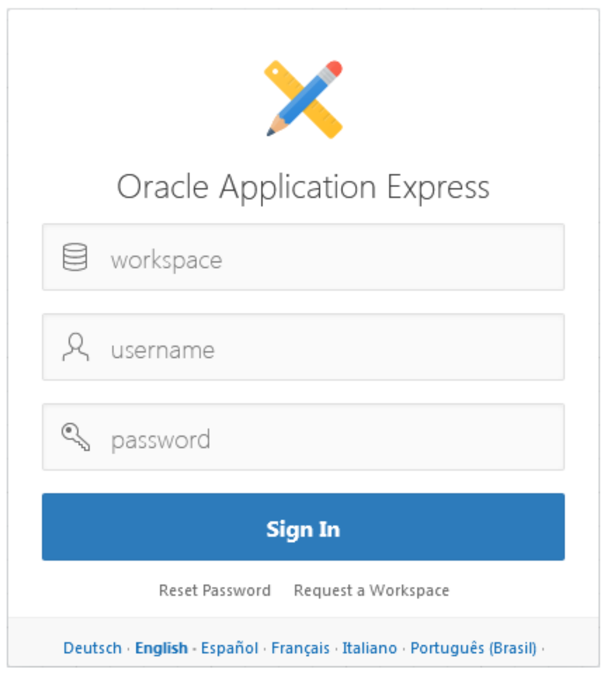


Figura 3.5: Dinámica de un request con el *mod_plsql* [88]

La descripción de esta dinámica es la siguiente:

1. El *web server* Apache (que atiende en host.dominio:7778) recibe un request con el indicador pls, y lo delega al *mod_plsql*
2. El *mod_plsql* prepara el request a la base, obteniendo del DAD los datos de conexión a la base correspondiente a la entrada apex51
3. El *mod_plsql* se conecta a la base e invoca el procedimiento `f`, pasándole el parámetro `p`
4. El procedimiento `f` parsea e interpreta el parámetro `p`, obteniendo múltiples parámetros, y genera el contenido de la página HTML que será la respuesta, eventualmente con referencias a imágenes de *filesystem*
5. La página HTML retorna al *mod_plsql*
6. El *mod_plsql* mantiene en *filesystem* el contenido que pueda ser cacheado, y eventualmente obtiene del *filesystem* las imágenes allí almacenadas
7. El Apache envía la respuesta al *browser* incluyendo, si las hubiera, las imágenes referenciadas

A su vez, la página HTML que se obtiene como respuesta, usualmente contiene links a otras páginas APEX para ser obtenidas por *HTTP GET* y *HTML Forms* para interactuar con la aplicación enviando datos a través de *HTTP POST*. Por ejemplo, si accedemos a `https://apex.oracle.com/pls/apex/` y revisamos el código fuente de la página obtenida (luego de la redirección al login), veremos algo como lo que se muestra en la Figura 3.6.




```
<form action="wwv_flow.accept" method="post" name="wwv_flow" id="wwvFlowForm" novalidate>
   <input type="hidden" name="p_flow_id" value="4550" id="pFlowId" />
   <input type="hidden" name="p_flow_step_id" value="1" id="pFlowStepId" />
   <input type="hidden" name="p_instance" value="8999145840213" id="pInstance" />
   ...
   <label for="F4550_P1_COMPANY" id="F4550_P1_COMPANY_LABEL">Workspace</label>
   <input type="text" id="F4550_P1_COMPANY" name="F4550_P1_COMPANY" placeholder="Workspace"
      required value="" size="40" maxlength="2000" />
   ...
   <label for="F4550_P1_USERNAME" id="F4550_P1_USERNAME_LABEL">Username</label>
   <input type="text" id="F4550_P1_USERNAME" name="F4550_P1_USERNAME" placeholder="Username"
      required value="" size="40" maxlength="2000" />
   ...
   <button onclick="apex.submit({request:&#x27;LOGIN_BUTTON&#x27;});"
      type="button" id="B2320055005809445564">Sign In</button>
   ...
   <li><a href="f?p=4550:1:8999145840213:::::&p_lang=en">English</a></li>
   <li><a href="f?p=4550:1:8999145840213:::::&p_lang=es">Español</a></li>
...
</form>
```

Figura 3.6: Ejemplo de código fuente de una página APEX

Es interesante notar que los parámetros que se pasan posicionalmente a través del parámetro `p` en el caso de los links, se pasan a través de parámetros ocultos en el caso de un *HTML Form*. En el código anterior se hacen evidentes los cambios en la terminología de APEX, de los cuales advertíamos más arriba. En particular, el parámetro `p_flow_id` representa el número de la aplicación, el parámetro `p_flow_step_id` representa el número de página, y el parámetro `p_instance` representa un identificador de sesión. Los parámetros que no están ocultos son propios de cada aplicacón, y así, el parámetro `F4550_P1_COMPANY`

representa el *Workspace* al cual el usuario se quiere loguear, que se seteará como un item llamado `P1_COMPANY` en la página 1 de la aplicación 4550 (en las secciones siguientes entraremos en estos detalles).

Cuando un usuario realiza un *HTTP GET* solicitando una página, el *web listener* utiliza una sesión del *pool* de sesiones de la base de datos para ejecutar un procedimiento `F` pasándole el parámetro `P` (entre otros) con la información obtenida de la URL. El procedimiento `F` se encarga de recuperar los metadatos relacionados a la página y generar el código HTML, después de pasar los controles de autenticación y autorización. Internamente, este proceso se conoce como *Page Rendering* o *Show Processing* y es manejado por un procedimiento llamado `show` dentro de un package de llamado `wwv_flow`.

De manera similar, cuando un usuario está en una página que contiene un *HTML Form* y realiza un *Submit*, se ejecuta un *HTTP POST* con la acción `wwv_flow.accept`, y el *web listener* utiliza una sesión del pool de sesiones de la base de datos para ejecutar un procedimiento `WWV_FLOW.ACCEPT`. Este procedimiento se encarga de recuperar la metadata relacionada a las validaciones, cómputos y procesos que se deben realizar. Internamente, este proceso se conoce como *Page Processing* o *Accept Processing*. A continuación de un *HTTP POST* se puede disparar automáticamente un *HTTP GET*.

La Figura 3.7 muestra el flujo del procesamiento de las páginas en cada uno de los dos casos.

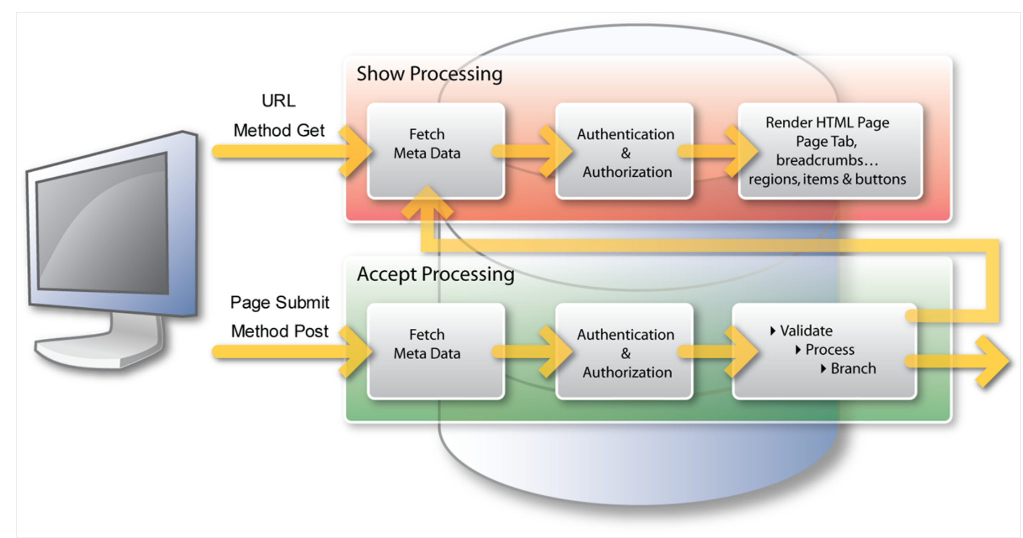


Figura 3.7: Procesamiento de páginas en APEX [89]

El costo del procesamiento de una página en APEX (incluyendo tareas

como leer la metadata, controlar autenticación y autorización, y generar el HTML) se mide normalmente en fracciones de segundo, y la mayor parte del tiempo que implica generar una página en la computadora del cliente se consume en las sentencias sobre los datos de usuario (e.g. `SELECT`, `INSERT`), latencia de la red, y procesamiento en el *browser* del cliente. APEX no consume ningún recurso del RDBMS (como los que resultarían de mantener una conexión para cada usuario) excepto cuando está procesando una página.

Lo presentado hasta aquí describe conceptualmente el proceso, si bien puede haber ejemplos más complejos por incluir más cantidad de inputs. Por ejemplo, una funcionalidad que permita buscar un cliente y ver sus datos, podría comenzar en una página de búsqueda de cliente, con campos de texto para buscar por nombre, apellido o cédula y un botón "Buscar". Una vez que el cliente ingresa un texto en alguno de estos campos y oprime el botón "Buscar", dentro del proceso de *Submit* se ejecuta un proceso de la página que ejecuta una consulta en la base de datos y presenta una grilla donde en cada fila aparece la cédula, nombre, apellido y un botón "Ver datos", de los clientes que hayan satisfecho la condición de búsqueda. Al presionar alguno de los botones "Ver datos", se realiza un *Submit* a otra página pasándole el identificador del cliente, y permite que se realice una consulta de los datos del cliente, como dirección, teléfono y fecha de nacimiento. Estos datos, a su vez, podrían cargarse en la página en campos de texto dentro un *HTML Form*, lo que permitiría modificarlos.

Cualquier investigación posterior sobre la vista de procesos, debería incluir un análisis detallado del paquete `wwv_flow`, que en la documentación de Oracle se conoce como *APEX Engine*.

3.5. Vista de desarrollo

Recapitulando lo visto en la Sección 3.2, es fácil imaginar que el desarrollo en APEX deberá comenzar con la creación de un *Workspace* por parte de un administrador, la creación de usuarios desarrolladores, la asociación del *Workspace* a esquemas de la base de datos sobre los que tendrá permisos, y la asignación de permisos a los usuarios desarrolladores. Una vez creado el *Workspace* y los usuarios, y asignados los permisos, un usuario desarrollador podrá crear una aplicación, asignarle un nombre y definir sus propiedades generales como tema, sistemas de autenticación y autorización, imagen para

un logo y URLs. Una vez creada la aplicación se pueden crear las páginas, y en cada página se pueden crear varias regiones, donde se ubican ítems y botones. Todo esto se puede complementar con la creación de menús y barras de navegación con links a las páginas, así como procesos para permitir cargar los valores de los ítems a partir de la base de datos, modificar la base de datos a partir de los valores de los ítems, o pasar de una página a otra pasando ítems como parámetros. En esta sección veremos cómo se puede realizar el desarrollo de una aplicación completa con base en estos elementos constructivos.

Un análisis más detallado de los elementos constructivos y sus relaciones puede obtenerse directamente a partir de las vistas del diccionario de datos de APEX, especialmente de las siguientes:

- `apex_workspaces`
- `apex_applications`
- `apex_application_pages`
- `apex_application_page_regions`
- `apex_application_page_buttons`
- `apex_application_page_items`

La vista de desarrollo de esta sección describe la organización estática del software en su ambiente de desarrollo. En el caso de una aplicación para usuarios finales, queda bien claro lo que significa una vista de desarrollo. En el caso de una aplicación que es un IDE para desarrollar aplicaciones, la expresión "vista de desarrollo" puede tener dos acepciones:

1. La vista que presenta la forma de utilizar el IDE, es decir, la forma de desarrollar aplicaciones en APEX
2. La vista que presenta la forma como está desarrollado APEX como producto

Hasta cierto punto, estas dos formas de interpretar la vista de desarrollo se parecen, ya que el propio IDE de APEX es un conjunto de aplicaciones APEX. Sin embargo, el desarrollo de una aplicación APEX para usuarios finales utilizando el IDE difiere del desarrollo del propio IDE fundamentalmente en que este último implica el desarrollo del *web listener*, del código PL/SQL de APEX, el mantenimiento de su modelo de datos y el mantenimiento de los datos que permiten generar la interfaz de usuario.

En esta sección se presenta una vista de desarrollo siguiendo la primera acepción. En la sección 3.6 se presentan algunas consideraciones sobre la vista de desarrollo siguiendo la segunda acepción.

Como hemos visto, APEX soporta que varias empresas o departamentos desarrollen sus aplicaciones de forma independiente. En la Figura 3.8 se puede visualizar cómo tres departamentos tienen diferentes aplicaciones sobre la misma base, algunas trabajando sobre esquemas totalmente independientes, y otras utilizando algún esquema en común. El administrador de una instalación de APEX, al crear un *Workspace*, define sobre qué esquemas tendrá permisos. De esta forma, el *Workspace* se convierte en una *Virtual Private Database* donde los diferentes departamentos pueden mantener sus aplicaciones y datos privados. A partir de este punto nos enfocaremos en el desarrollo de una aplicación, asumiendo que tenemos un esquema con los objetos y datos necesarios.

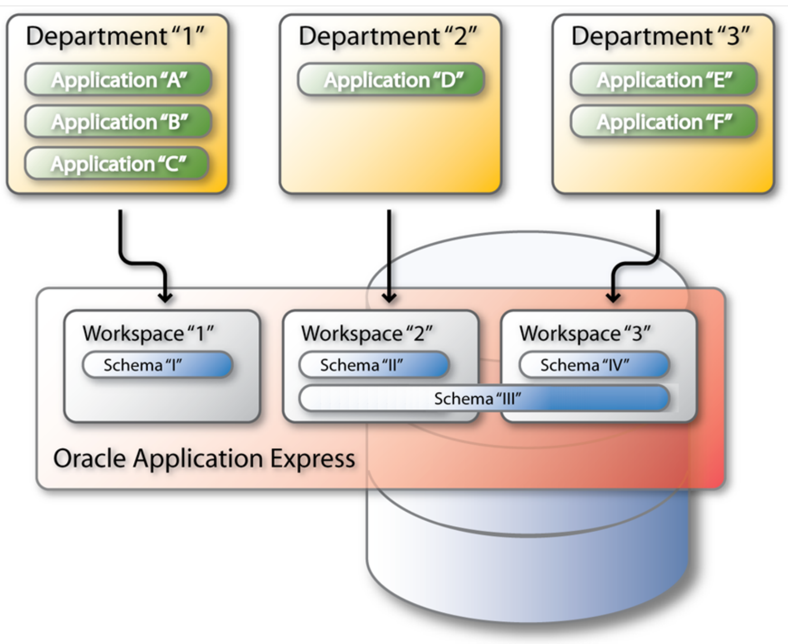


Figura 3.8: Relación entre *Workspaces*, aplicaciones y esquemas [89]

Una aplicación APEX, además de interactuar con los objetos locales de los esquemas a los que tiene acceso, podría interactuar con objetos remotos a través de *Database Links* y con *Web Services*. Para simplificar, nos enfocaremos solamente en la utilización de objetos locales.

La instalación de APEX crea un *Workspace* por defecto (llamado *Internal Workspace*) y deja disponibles múltiples aplicaciones que juntas conforman el

IDE. Entre ellas, se destacan las que se presentan en la Figura 3.9.

```
SQL> select id, name from wwv_flows
     where id in (4000, 4350, 4500, 4550);

Id         Name
---------- ----------------------------------------
      4000 Oracle APEX AppBuilder
      4350 Oracle APEX Workspace Administration
      4500 Oracle APEX SQL Workshop
      4550 Oracle APEX Login
```

Figura 3.9: Principales aplicaciones del IDE de APEX

No entraremos en detalles de la aplicación 4350, más allá de mencionar que es la que utiliza el usuario *Instance Administrator* para crear y administrar *workspaces*, incluyendo la creación de usuarios de tipo *Workspace Administrator*, *Developer* y *End User*. Después de creado el *Workspace*, la aplicación 4550 permite el login al *Workspace*, tal como se mostró en la Figura 3.6. Después del login en la aplicación 4550 se presenta la página principal del IDE (página 1000 de la aplicación 4500) que se muestra en la Figura 3.10.

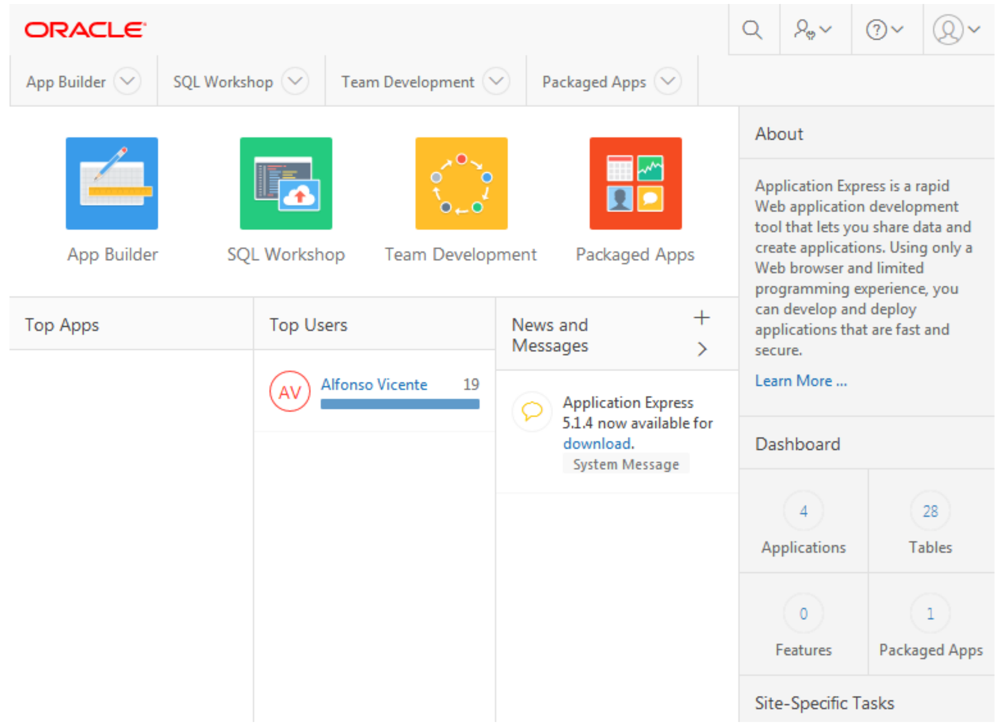


Figura 3.10: Página principal del IDE de APEX

Típicamente, una aplicación se desarrolla utilizando la aplicación *App Builder*. La aplicación *SQL Workshop* permite ejecutar consultas *ad hoc* así como escribir, mantener y ejecutar scripts SQL y PL/SQL, lo que puede ser útil cuando se desarrolla una aplicación nueva, o necesario cuando no se tiene acceso directo a la base, como en el caso de tener un entorno de APEX como *Software As A Service* (SaaS). No diremos nada de las aplicaciones *Team Development* y *Packaged Apps*.

Dentro de la aplicación *App Builder*, se puede crear una nueva aplicación así como modificar una aplicación existente. Hay funcionalidades para editar las propiedades generales de la aplicación, lanzar una ejecución de la aplicación, exportar e importar una aplicación o una página, mantener *Shared Components* (como listas de valores, menús, barras de navegación, y sistemas de autenticación y autorización) y mantener las páginas de la aplicación.

En la Figura 3.11 se muestra el primer paso de la creación de una página. Sobre las interfaces, solo mencionaremos que existe la posibilidad de generar una página para interfaz para escritorio y para móviles, con lo cual una aplicación APEX puede ser *responsive*, detectando cuando debe mostrar una interfaz para escritorio o para móvil, y adaptando el contenido al tamaño de la ventana del *browser* o pantalla del dispositivo. El tipo de página permite que se creen regiones y componentes preestablecidos en el mismo momento de la creación de la página, pero siempre es posible partir de una página en blanco y llegar al mismo resultado editándola. Los componentes preestablecidos pretenden ahorrar tiempo, existiendo cinco tipos de *reports* y nueve tipos de *forms* para los casos más usuales.

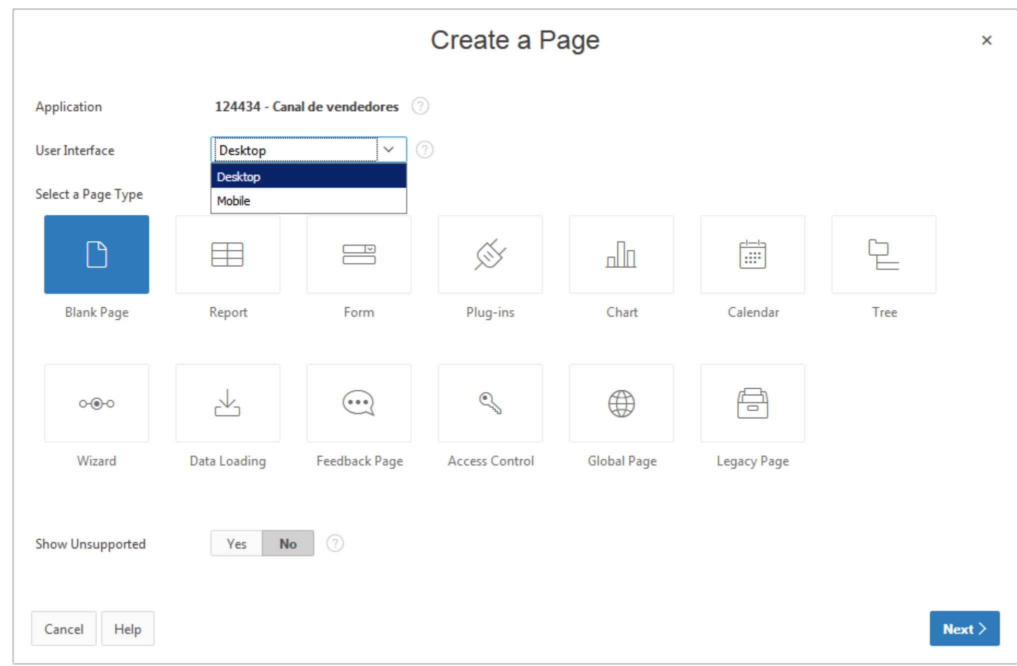


Figura 3.11: Creación de una página en APEX

Una vez creada una página puede editarse a través del *Page Designer*. En esta página es en la que se invertirá más tiempo al desarrollar una aplicación utilizando APEX, ya que consolida todo lo que puede hacerse en una página (definir regiones en blanco o de tipos preestablecidos como *reports* y *forms*, ítems y botones dentro de las regiones, acciones dinámicas y procesos). Se presenta una vista del *Page Designer* en la Figura 3.12.

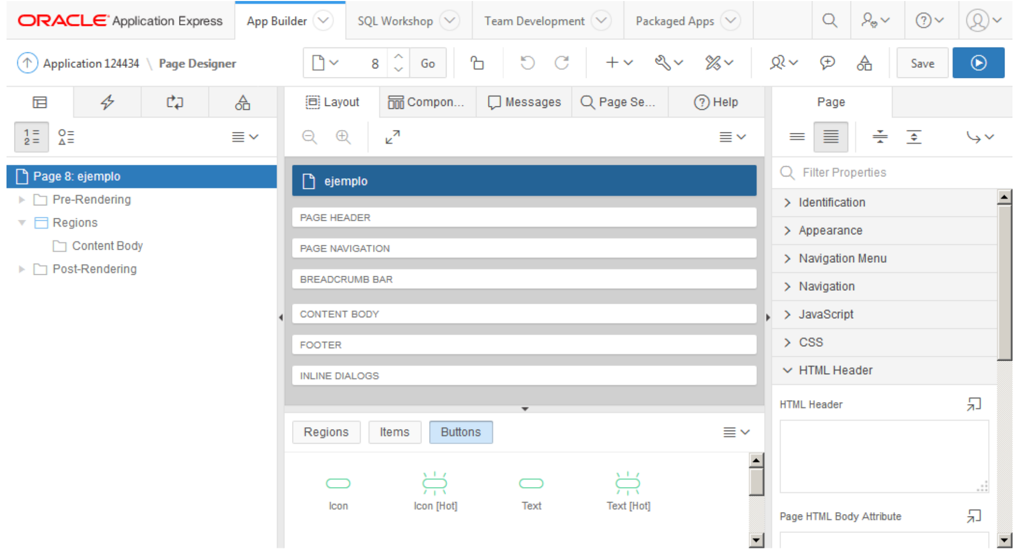


Figura 3.12: Vista del *Page Designer*

Una página debe tener al menos una región para poder tener contenido, pero típicamente una página tiene varias. En la Figura 3.11, cuando el IDE ofrece crear una página de tipo *Report* o *Form*, significa que creará una página con una o más regiones de tipo *Report* o *Form*, dependiendo del tipo de *Report* o *Form* que se seleccione en un paso subsiguiente. Podemos lograr el mismo resultado agregando regiones a una página en blanco. En la Figura 3.12 se ven los elementos (regiones, ítems y botones) que se pueden agregar a las diferentes secciones preestablecidas de la página, lo que se puede lograr arrastrando y soltando o mediante un menú contextual sobre el elemento. Veremos los casos más simples de creación de regiones *Report* y *Form*, notando cómo se traducen estos componentes preestablecidos en términos de los componentes básicos.

Al crear una región de tipo *Report*, se ofrecen varias alternativas de tipo, aunque se diferencian fundamentalmente en las funcionalidades adicionales que se tienen sobre el reporte. Veremos el caso de un *Interactive Report*, por ser uno de los más ricos en funcionalidad. En la Figura 3.13 se puede ver la apariencia del *Page Designer* trabajando sobre una página en la que se creó un *Interactive Report*. El dato requerido para el reporte es una consulta SQL, a partir de cuya proyección se crean automáticamente ítems que sirven para generar el reporte.

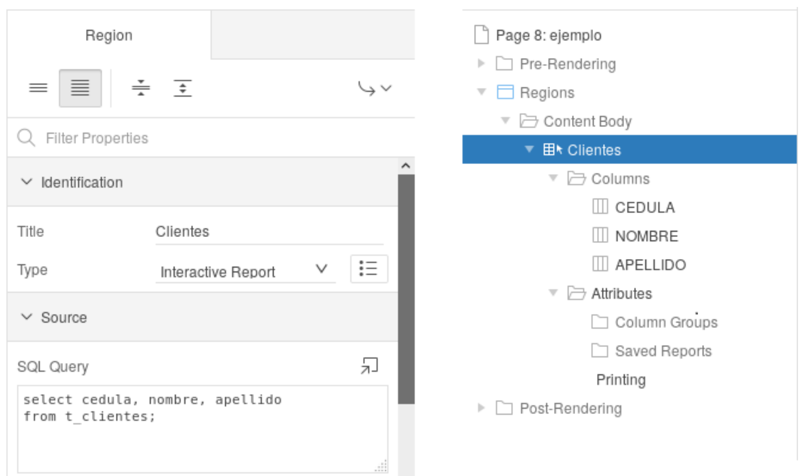


Figura 3.13: Creación de un *Interactive Report*

Al ejecutar un *Interactive Report* se ejecuta la consulta, los datos se obtienen de forma paginada, y la primer página retorna al cliente, que tiene a disposición una amplia gama de controles para realizar personalizaciones sobre el reporte, como mostrar, ocultar y ordenar columnas, ordenar las tuplas, filtrar datos mediante predicados, realizar funciones de agregación,

realizar cortes de control, o resaltar datos mediante predicados. La mayor parte de estas acciones se implementan mediante un *HTTP POST* a un package `wwv_flow.ajax`, que permite realizar actualizaciones parciales de una región mediante AJAX. Dejaremos el análisis de esta funcionalidad fuera de alcance, pero vale mencionar que permiten que un reporte sea dinámico y personalizable por el usuario, sin costo de desarrollo. Todas estas personalizaciones se pueden guardar como versiones del reporte con nombre (*named reports*) para su uso posterior, y una de ellas se puede predefinir como la versión *default*. Distintos usuarios pueden tener distintas personalizaciones del mismo reporte, realizadas por ellos mismos. A partir de un reporte de todas las ventas del último mes, podrían haber personalizaciones que muestren solo las ventas de alguna sucursal, sumas de totales de ventas, o colores de las tuplas diferenciados según el monto de la venta. En la Figura 3.14 se puede apreciar la interfaz de un *Interactive Report*.

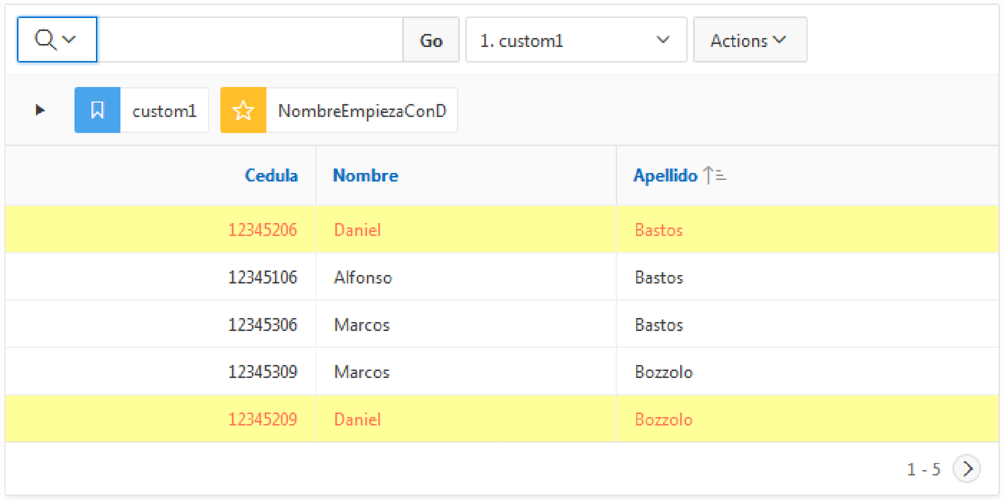


Figura 3.14: Vista de un *Interactive Report*

Al crear una página con una región de tipo *Form*, el *Create Page Wizard* ofrece varios tipos disponibles de *Form*, como se puede ver en la Figura 3.15. Cada una de estas opciones determinará un flujo diferente en el *wizard*, solicitando los datos necesarios en cada caso, y terminará creando una página o dos, con todos los controles necesarios para lograr una interfaz y una navegación muy básicas, pero funcionales. Analizaremos la opción de *Report with Form on Table*, por ser relativamente simple y por incluir una navegación con *branches*, que es la base de la navegación de una aplicación compleja de tipo *data-driven*. El objetivo será crear una página con un reporte que permita

buscar en los datos de una tabla, y navegar a otra página con un formulario que permita modificarlos.

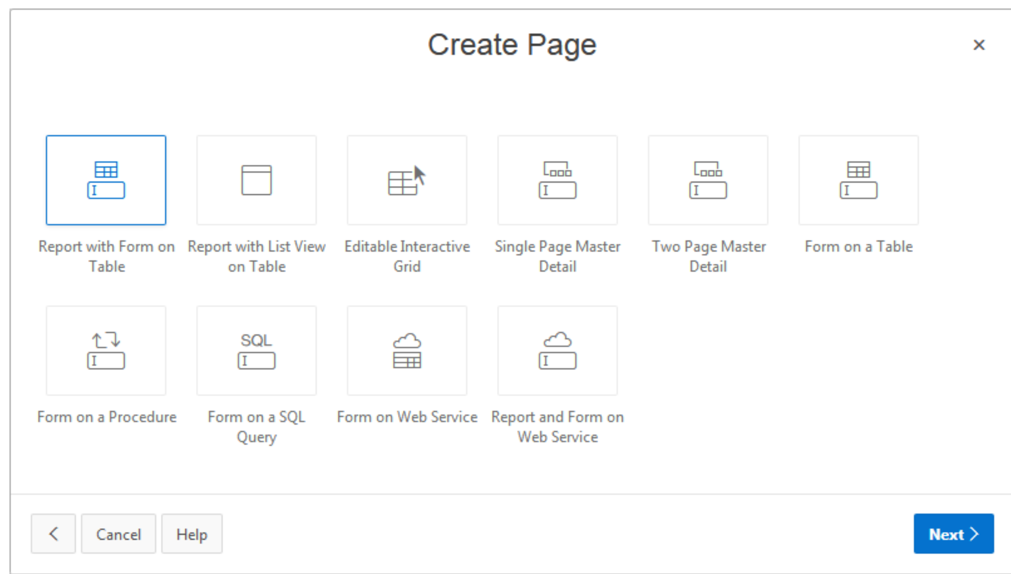


Figura 3.15: Opciones de Form en el *Create Page Wizard*

Solo ingresando el nombre de la tabla y unos pocos datos más en el *wizard*, se logran un par de páginas funcionales y navegables entre sí, como se pueden ver en la Figura 3.16. A la izquierda se muestra una sección de la página con el reporte, que incluye un botón *"Create"* para crear una nueva tupla en la tabla, y un *link* en cada tupla (en la imagen del lápiz amarillo) para modificar una tupla existente. Al presionar cualquiera de los controles (el botón o uno de los *links*) se pasa a la página cuya sección se muestra a la derecha. En el caso que se haya llegado a partir de un *link* de modificar, se cargan los datos de la tupla correspondiente; y en el caso de que se haya llegado a partir del botón de crear, todos los campos aparecen vacíos. Desde la página del *Form* es posible crear una tupla nueva, modificar una tupla existente o eliminar una tupla existente. Después de oprimir el botón de aplicar los cambios, se regresa a la página del reporte. A partir de este esqueleto básico funcional, es posible personalizar tanto el comportamiento como la apariencia de múltiples maneras.

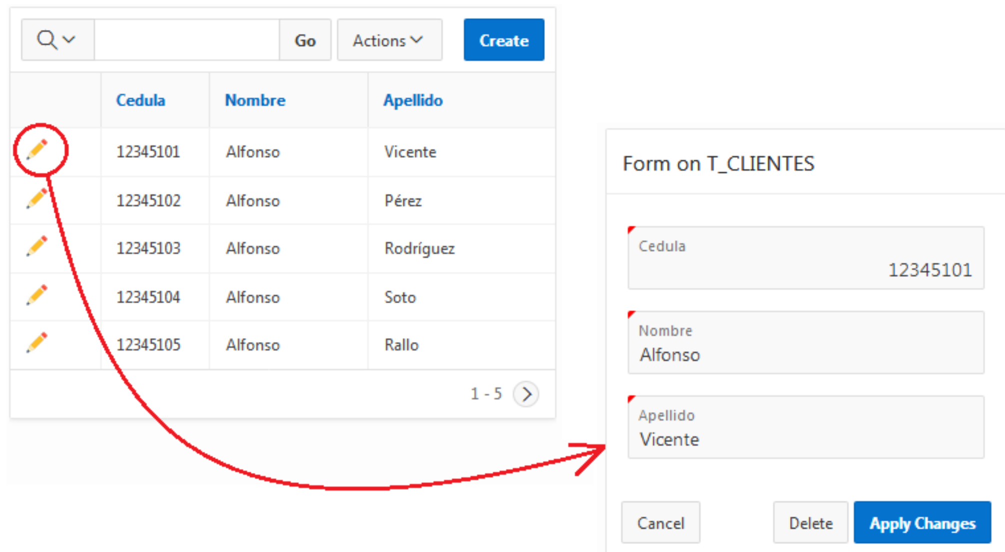


Figura 3.16: Funcionamiento del *Report with Form on Table*

Para comprender la navegación debemos recurrir al *Page Designer*. En el caso del ejemplo se crearon dos páginas: la página del reporte (como página 5) y la página del formulario (como página 6). En la página 5 del *Report*, se creó una *Link Column*, que tiene como destino la página 6 del *Form*, y que setea en el ítem `P6_ROWID` el valor del `ROWID` de la tupla en la que se oprimió el *link*. Los aspectos más relevantes de esta configuración se pueden ver en la Figura 3.17. Esta es la forma de navegar entre páginas de una aplicación, pasando datos de una página a otra.

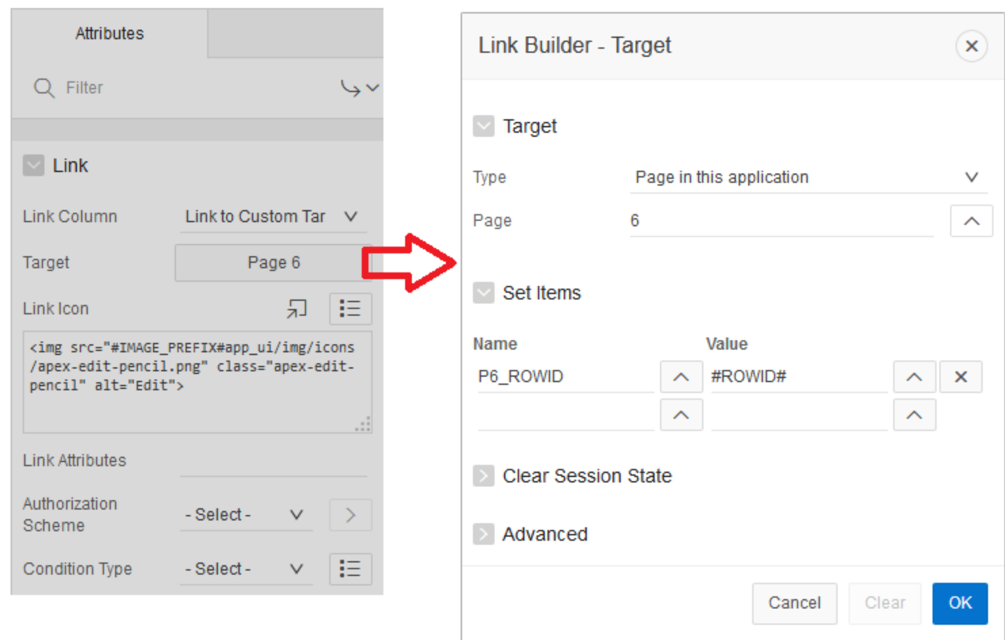


Figura 3.17: *Set Items* en el *Page Designer*

En la página 6 del *Form*, se crearon varios controles para completar el comportamiento deseado, como se puede ver en la Figura 3.18. Por un lado, en la región de *Rendering* se creó un proceso *Fetch Row from T_CLIENTES* del tipo predefinido *Automatic Row Fetch*, que es el que permite ejecutar el `SELECT` para obtener la tupla de `T_CLIENTES` que se pasó desde el *Report* de la página 5, si es que se llegó a través del *link*. Por otro lado, en la región de *Processing* se crearon dos procesos: *Process Row of T_CLIENTES*, del tipo predefinido *Automatic Row Processing (DML)*, que implementa las sentencias `INSERT`, `UPDATE` y `DELETE` en el *Submit*; y *Reset Page*, del tipo predefinido *Clear Session State*, que limpia los valores de los ítems en el *Submit* de forma que cualquier subsiguiente invocación de la página no tenga ítems con valores cargados en una invocación anterior. También se creó un *branch* llamado *Go To Page 5* para que después de un *Submit* se vuelva a la página 5 donde está el *Report*.

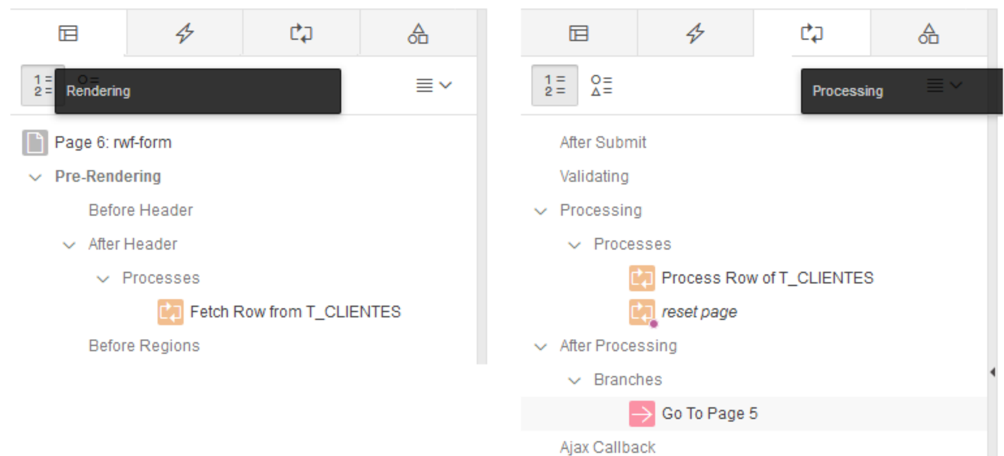


Figura 3.18: Controles del Form en el *Page Designer*

Todos estos controles fueron creados por el *wizard* de forma automática, pero se podría haber logrado el mismo resultado a partir de páginas en blanco, creando cada una de las páginas, regiones, botones, ítems y procesos. Desde el punto de vista del desarrollo, contar con estas opciones prediseñadas permite lograr muy rápidamente un prototipo que se puede hacer evolucionar para convertirse en la aplicación deseada, aunque una aplicación real siempre requerirá un considerable trabajo posterior para modificar tanto la apariencia como el comportamiento.

Otra funcionalidad a destacar es la posibilidad de definir de forma declarativa *dynamic actions*, a partir de un gran número de acciones predefinidas que generan de forma automática código JavaScript o AJAX, que puede ser ejecutado de forma sincrónica (bloqueando el *browser*) o asincrónica. Es posible crear una *dynamic action* a partir de un ítem o un botón, y un evento (por ejemplo, cuando se hace click en un botón o se cambia el valor de un ítem), y aún es posible según el caso agregar un predicado al evento (por ejemplo, si el nuevo valor de un ítem no es `null`). Una vez creada la *dynamic action*, se pueden crear múltiples acciones a ejecutarse en secuencia en el caso en que el predicado sea verdadero y en el caso que sea falso. Estas acciones pueden incluir mostrar u ocultar una región entera; mostrar, ocultar, habilitar o deshabilitar un ítem o botón; setear el valor de un ítem; refrescar otra región con un reporte; y muchas más alternativas predefinidas, así como ejecutar código JavaScript ingresado por el usuario.

Típicamente, el proceso de desarrollo incluye el diseño de menús, *breadcrumbs*, *popups* con listas de valores, acciones dinámicas que permitan

ocultar o mostrar otros componentes, *links* y botones para lograr las navegaciones deseadas y personalización de la apariencia.

3.6. Consideraciones sobre el IDE

En la sección anterior se mencionó que haríamos algunos comentarios sobre la vista de desarrollo siguiendo la segunda acepción de una "vista de desarrollo" para el caso de un IDE, es decir, la vista que presenta la forma en que está desarrollado APEX como producto. Es posible ver a APEX como una combinación del *web listener*, la *APEX Engine* y el conjunto de aplicaciones APEX que conforman el IDE.

Si bien no disponemos del código del *web listener*, para analizar su funcionamiento básico podemos considerar como alternativa la distribución *Open Source* del *Apache PL/SQL Gateway Module* o *mod_owa*. Si bien este módulo no tiene todas las funcionalidades del *web listener* actual de APEX, sí incluye las funcionalidades básicas que permiten realizar *HTTP GET* y *HTTP POST* para ejecutar un *stored procedure* `OWA.GET_PAGE`, una forma estándar de obtener páginas, así como *upload* y *download* de archivos. Este desarrollo debe resolver en lenguaje C la integración estándar como un módulo de Apache para los métodos *HTTP GET* y *HTTP POST*, la lectura de un archivo de configuración, la conexión a la base, el mantenimiento de un *pool* de conexiones, y la ejecución de *stored procedures* que deben desarrollarse en paralelo en el RDBMS [90].

La *APEX Engine* está implementada en el package `wwv_flow`, y tiene dos procedimientos fundamentales: `show`, encargado del *Page Rendering* o *Show Processing* o gestión de los *HTTP GET*, y `accept`, encargado del *Page Processing* o *Accept Processing* o gestión de los *HTTP POST*. En los *HTTP Requests* vemos la llamada directa a `accept`, como se vio en la Figura 3.6, pero no vemos la llamada a `show`, y vemos en cambio una llamada a un procedimiento `f`. Esto ocurre porque `f` es un *wrapper* del procedimiento `show`, al que termina llamando. Como se vio en la Figura 3.7 el procedimiento `accept` debe procesar las validaciones, procesos y *branches*, y eventualmente redirigir a otra página por medio de un `show`. La fase final y más compleja del procedimiento `show`, es la del *rendering* de la página HTML. Para esto se realizan llamadas anidadas a través de funciones que retornan `VARCHAR2` para construir cada uno de los elementos de la jerarquía, que terminan ejecutando

los procedimientos `sys.htp.p` y `sys.htp.prn` que permiten agregar líneas al *HTTP buffer*, y cuando se completa la construcción dinámica de la página se llama a `apex_application.stop_apex_engine` para cerrar el *HTTP buffer*, y finalmente se llama a la función `wwv_flow_response.get_response`, que termina ejecutando un `sys.htp.get_page`. De esta forma, el funcionamiento de la *Apex Engine* descansa sobre el package `sys.htp`, preexistente en la base Oracle, aunque *a priori* no parece que realice una función indispensable para el funcionamiento de la *Apex Engine*, sino más bien que simplifica la tarea por ser un conjunto de funciones utilitarias diseñadas especialmente para trabajar con un *buffer* de HTTP por sesión. La característica más importante de las funciones del package `sys.htp`, es que permiten generar un HTML en un buffer que se puede escribir de forma posicional, con lo cual es posible escribir más líneas en el *header* después de haber escrito algunas líneas del *body*, o recuperar y reescribir regiones de una página HTML lo que facilita la utilización de tecnologías como AJAX.

El desarrollo de las aplicaciones del IDE debe haber tenido una gran complejidad. Por un lado porque hay aplicaciones y páginas muy complejas por todo lo que integran, como la página del *Page Designer*. Por otro lado porque si bien es posible que estas aplicaciones se puedan mantener con el propio IDE, es seguro que muchas nuevas funcionalidades se deban preparar sin el auxilio del IDE. También queda por fuera del IDE todo el desarrollo de las múltiples rutinas en JavaScript, el mantenimiento de las CSS y los *tags* de HTML que están distribuidos por las tablas. El mantenimiento del esquema de base de datos, y del código PL/SQL de los múltiples *packages* y *procedures* representa un desafío. En términos generales, es de esperar que el desarrollo y mantenimiento de un IDE como el de APEX sea un trabajo difícil.

3.7. Vista de casos de uso

Para ilustrar y complementar la información de las cuatro vistas anteriores, se presentan a continuación unos pocos casos de uso, que conforman esta quinta vista.

Desarrollaremos en primer lugar el caso de uso genérico de un *HTTP GET*, cuyo diagrama se puede ver en la Figura 3.19.

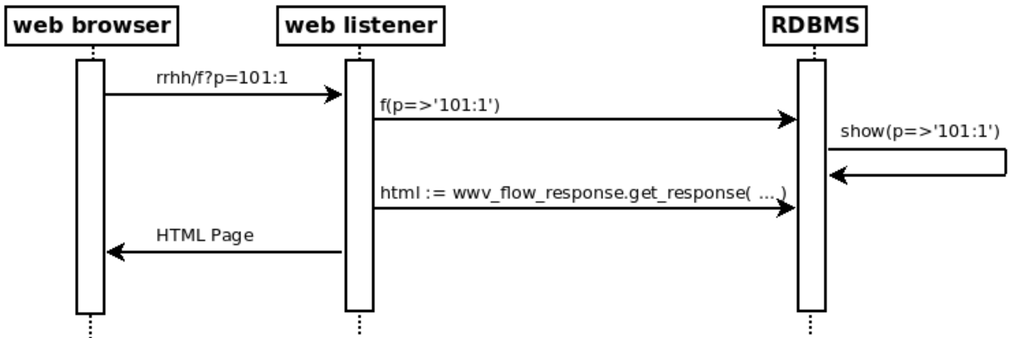


Figura 3.19: Caso de uso genérico de un HTTP GET

La descripción del caso de uso es como sigue:

1. El *web browser* envía un *HTTP Request* con método *HTTP GET* a una URL donde atiende un *web listener* de APEX, pasándole una entrada correcta del DAD, y llamando al procedimiento `f` con el parámetro `p=101:1`
2. El *web listener* se conecta a la base indicada por la entrada `rrhh` del DAD, y ejecuta el procedimiento `f` pasándole (entre otros) el parámetro `p=>'101:1'`
3. En el RDBMS el procedimiento `f` ejecuta el procedimiento `wwv_flow.show`, pasándole (entre otros) el parámetro `p='101:1'`. Este procedimiento ejecuta, en cascada, varios otros procedimientos que realizan el *rendering* de los componentes (regiones, ítems, botones, procesos) de la página 1 de la aplicación 101, agregan el resultado como una página HTML y lo dejan disponible para la sesión en un *buffer* de HTTP a través del *package* `sys.htp`.
4. El *web listener*, al terminar la ejecución del procedimiento `f`, ejecuta la función `wwv_flow_response.get_response` y recibe como respuesta la página 1 de la aplicación 101
5. El *web listener* prepara el *HTTP Response* con la página 1 de la aplicación 101 y se lo retorna al *web browser*

La llamada al procedimiento `wwv_flow.show`, que se observa como un único paso en la Figura 3.19 resuelto por el RDBMS visto como una caja negra, se puede desagregar en una vista de caja blanca como se observa en el diagrama de casos de uso de la Figura 3.20. En este caso se decidió mostrar solo algunos de los flujos posibles. Se

omitieron varias llamadas, algunas de ejecución segura como `wwv_flow_meta_data.fetch_step_info` o `wwv_flow.run_before_header_code`, y otras que son de ejecución posible como: `wwv_flow_ppr_util.run_process` o `wwv_flow_region_native.render_classic_report`. En un caso de uso genérico del *Page Rendering* de una página cualquiera, no hay un flujo principal como tal, y las llamadas entre procedimientos y funciones dependerán de los componentes de la página. De encontrarse un error en cualquier punto, se ejecuta un `wwv_flow.stop_apex_engine` y se retorna una página con la descripción del error.

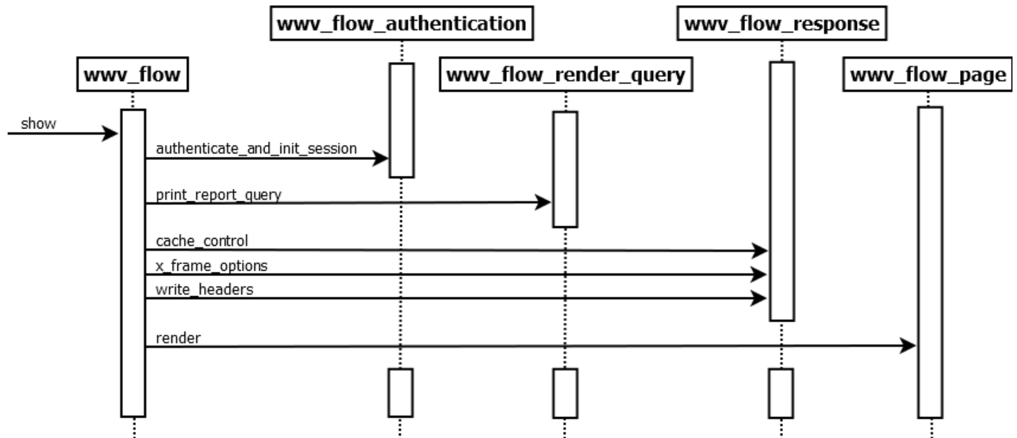


Figura 3.20: Caso de uso genérico del wwv_flow.show

A su vez, la última llamada a `wwv_flow_page.render` se podría analizar como otro caso de uso complejo donde se generan llamadas nuevamente al package `wwv_flow`, como `wwv_flow.paint_buttons`, y de allí a `wwv_flow_button.render`. En el transcurso de todas estas llamadas se generan porciones de código HTML que se van insertando en el *buffer* del *package* `sys.htp`.

3.8. El caso del manejo de sesión

En esta sección se presenta específicamente el caso de uso del manejo de la sesión, debido a la importancia que reviste.

HTTP es un protocolo sin estado, y aunque desde HTTP 1.1 las conexiones son persistentes, las *HTTP cookies* (en adelante simplemente *cookies*) siguen siendo el mecanismo estándar para mantener el estado de las sesiones de los usuarios [91, 92]. Para poder mantener un estado se recurre al concepto de sesión, una construcción lógica que permite implementar la persistencia de valores entre los sucesivos *HTTP Requests* que realice un mismo usuario. Para

identificar las sesiones usualmente se genera un identificador único llamado *session ID* [93].

Cada cliente logueado debe poder mantener la metáfora de una sesión en la aplicación, y esto se logra por medio de una *cookie* que mantiene un *session ID* que se obtiene durante un proceso estándar de login. Con este *session ID* se determina si el cliente está autenticado, y en caso de no estar autenticado cuando trata de acceder a una página que requiere autenticación, se redirecciona a la página de login propia de la aplicación (ignoraremos los casos de autenticación donde es posible interoperar con otros sistemas como por ejemplo un *Single Sign-On* a través de protocolos como *OpenID Connect* y *OAuth*).

La descripción de este proceso es la siguiente. Un cliente accede a una página de login de APEX, a través de un *HTTP GET* a una URL que termina con algo como `f?p=1234:101`. En este ejemplo se trata de la página 101 (de login) de la aplicación 1234. Esto retorna una página con un *HTML Form* donde se ingresará usuario y contraseña, y se enviará por *HTTP POST*, incluyendo los inputs como `p_flow_id=1234`, `p_flow_setp_id=101`, `p_instance=123456789` (que ahora no referencia a una sesión vigente), `F1234_P101_USERNAME=JPEREZ` y `F1234_P101_PASSWORD=MUYSECRETA`. Después de procesar la función de autenticación y resultar un login exitoso, se setea en el *HTTP Response* una *cookie* llamada `ORA_WWV_APP_1234` con un valor que internamente referencia a un identificador único, por ejemplo `ORA_WWV-tShYTadxgt_62LLi50fFn2u4`. El valor de esta cookie queda en la columna `COOKIE_VALUE` de la vista `WWV_FLOW_SESSIONS$`, y el valor asignado a la columna `ID` sirve para identificar la sesión. El *HTTP Response* es un *redirect* a la página principal, como `f?p=1234:1:7869946361491`, donde ya tiene asignado un identificador de sesión que sí referencia a una sesión vigente. En la Figura 3.21 se muestran las principales columnas de la vista dinámica `WWV_FLOW_SESSIONS$`, donde se puede apreciar que también se mantiene el `USERNAME` del usuario y un identificador `SECURITY_GROUP_ID` que referencia a un *Workspace* (a la columna `PROVISIONING_COMPANY_ID` de la tabla `WWV_FLOW_COMPANIES`), para asegurar el aislamiento de las aplicaciones de diferentes *Workspaces*. Tal vez lo más interesante de esta solución, es que APEX permite estar logueado en diferentes aplicaciones con diferentes usuarios, utilizando el mismo web browser, y manteniendo la sesión en todas ellas.

```
SQL> desc wwv_flow_sessions$

Name                         Null?    Type
---------------------------- -------- --------------
ID                           NOT NULL NUMBER
COOKIE_VALUE                 NOT NULL VARCHAR2(32)
SECURITY_GROUP_ID            NOT NULL NUMBER
CRYPTO_SALT                  NOT NULL RAW(32)
CREATED_ON                   NOT NULL DATE
MAX_IDLE_SEC                 NOT NULL NUMBER(5)
IDLE_TIMEOUT_ON              NOT NULL DATE
LIFE_TIMEOUT_ON              NOT NULL DATE
REMOTE_ADDR                           VARCHAR2(255)
ON_NEW_INSTANCE_FIRED_FOR             VARCHAR2(4000)
USERNAME                              VARCHAR2(255)
AUTHENTICATION_RESULT                 NUMBER
SCREEN_READER_MODE_YN                 VARCHAR2(1)
HIGH_CONTRAST_MODE_YN                 VARCHAR2(1)
SESSION_TIME_ZONE                     VARCHAR2(255)
SESSION_LANG                          VARCHAR2(5)
SESSION_TERRITORY                     VARCHAR2(255)
RAS_SESSIONID                         RAW(16)
WORKSPACE_USER_ID                     NUMBER
DEBUG_LEVEL                           NUMBER(1)
TRACE_MODE                            NUMBER(1)
TIMEOUT_COMPUTATION                   NUMBER

SQL> select ID, COOKIE_VALUE, SECURITY_GROUP_ID, USERNAME
     from wwv_flow_sessions$;

           ID COOKIE_VALUE     SECURITY_GROUP_ID USERNAME
------------- ---------------- ----------------- --------
7869946361491 ORA_WWV-t...n2u4                10 JPEREZ
```

Figura 3.21: Sesiones en la vista WWV_FLOW_SESSIONS$

En este ejemplo sencillo, se vio cómo se implementa un login mediante un *HTTP GET* a una página que contiene un *HTML Form*, cómo se envían los datos de esta página por *HTTP POST* y cómo se setea una *cookie* para mantener la sesión y se hace un *HTTP Redirect* para redirigir al cliente a una página determinada. Una vez logueado en la aplicación, el cliente típicamente navega por las diferentes páginas a través de links ubicados en menús, barras de herramientas, o dentro de las páginas (lo que supone un *HTTP GET*), y a través de las acciones de los botones y *branches* (lo que supone un *HTTP*

GET o un *HTTP POST* según lo que se indique). Un *branch* es una tarea que se ejecuta después de un *HTTP POST* o *Submit* (es decir, después de haber orpimido un botón) y que permite hacer un *HTTP Redirect* o ejecutar otro *Submit* a una página u otra siguiendo el código contenido en el *branch*, y en el caso del *Submit* eventualmente pasándole parámetros.

Si bien la *cookie* sirve para mantener identificada una sesión, todo el estado de la sesión se mantiene en la base de datos. El estado de una sesión se identifica por el *session ID* e incluye valores como el *workspace*, la aplicación, la página, el nombre del usuario, el idioma del *browser*, ítems de aplicación e ítems de página. APEX incluye la API `apex_util`, que tiene funciones para poder obtener el valor de un ítem en la sesión actual, como se ve en la Figura 3.22.

```
SQL> select item_name,
     apex_util.get_session_state(item_name) session_value
     from apex_application_page_items
     where application_id = 103 and page_id = 402;

ITEM_NAME            SESSION_VALUE
-------------------- --------------------
P402_CODIGO
P402_DESCRIPCION
P402_HABILITADO
P402_OBSERVACIONES
P402_FECHA
P402_USUARIO
```

Figura 3.22: Mantenimiento del estado de la sesión

La consulta de la Figura 3.22 muestra los ítems de página de la página 402 de la aplicación 103. Los valores no pueden ser obtenidos por fuera de APEX, ya que la función `apex_util.get_session_state()` solo retorna el estado de la sesión actual. Esta misma consulta sí retorna los de la sesión cuando es ejecutada en una aplicación APEX, lo que ejemplifica el proceso de recuperación del estado de la sesión.

3.9. Dos lecciones del análisis de APEX

De todo el análisis de APEX presentado en este capítulo, surgen dos lecciones importantes que conviene destacar con miras a realizar una propuesta

de arquitectura *RDBMS-only* en general.

La primera es que al utilizar APEX para desarrollar una aplicación, ésta tendrá una arquitectura *database-centric* como la propuesta por Koppelaars que se puede ver en la Figura 2.11; pero sin embargo se puede seguir o no el enfoque de ThickDB mostrado en la Figura 2.14. De hecho, la mayor parte de las opciones del IDE de APEX están orientadas a trabajar directamente sobre las tablas, y lo más usual en los cursos de Oracle donde los desarrolladores se pueden formar en esta tecnología, es trabajar directamente sobre las tablas en lugar de utilizar una o varias capas de PL/SQL que realicen operaciones más abstractas escondiendo la implementación en términos de sentencias SQL [94, 95]. Se puede argumentar que si la tecnología incluye más opciones para trabajar directamente sobre las tablas, si hace que trabajar de esta manera sea más fácil, y si la mayor parte de los ejemplos ofrecidos para formarse también inducen a trabajar de esta manera; no se generan las condiciones más propicias para que las aplicaciones construidas con APEX utilicen y se beneficien del enfoque de ThickDB.

La segunda es que APEX no tiene una frontera clara entre el comportamiento y la interfaz, al punto que los diferentes *tags* de HTML están distribuidos tanto por el código PL/SQL de la *APEX Engine* como por las tablas. Esto puede deberse a una decisión de diseño (o a la falta de una) que se remonte a sus humildes orígenes, donde la ausencia de un patrón de diseño que separara estos aspectos tal vez no resultaba un problema serio. Como sea que fuere, es seguro que la evolución de APEX representa actualmente un desafío para los desarrolladores de Oracle.

Estas dos lecciones permiten formular dos recomendaciones que serán tenidas en cuenta en la propuesta arquitectónica:

1. Facilitar el enfoque de ThickDB
2. Separar claramente la interfaz del comportamiento

En el siguiente capítulo se presenta una propuesta de arquitectura *RDBMS-only*. Para esta propuesta, el análisis de APEX de este capítulo resultó de mucha ayuda, si bien la arquitectura *RDBMS-only* difiere de la de APEX en varios aspectos.

Capítulo 4

Propuesta de la arquitectura RDBMS-only

> Con un simple cambio de énfasis puede cambiar todo: no solo nuestra comprensión, sino el problema mismo, su fertilidad y su significado, así como las perspectivas de una solución interesante.
>
> *El mito del marco común*
> *Karl R. Popper*

Este capítulo presenta la propuesta de la arquitectura *RDBMS-only*, que podría verse como un *"simple cambio de énfasis"* respecto de las arquitecturas de múltiples niveles tecnológicos y físicos. En su lugar, se propone una arquitectura de múltiples capas, pero siendo estas capas lógicas, dentro de un RDBMS.

Se presentará la arquitectura mediante dos acercamientos subsiguientes. El primero es un diseño general que resulta de integrar las recomendaciones de la Sección 3.9, pero abstrayéndose del problema del IDE. Este problema de integrar un IDE en la arquitectura, se aborda en el segundo acercamiento.

El diseño básico de la arquitectura se puede apreciar en la Figura 4.1, y es un enfoque de ThickDB que incluye una capa exterior donde se puedan definir *wrappers* para la interfaz. En este esquema los procedimientos de cada

capa solo tienen permitido conocer objetos e invocar procedimientos de la capa inmediatamente inferior, y eventualmente de su misma capa.

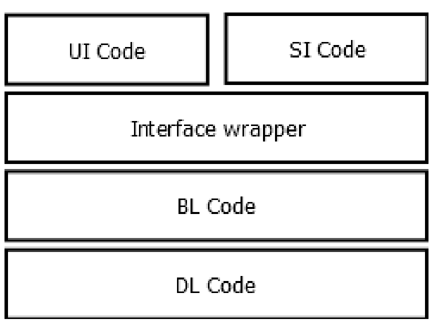


Figura 4.1: Primer acercamiento a la arquitectura

La capa indicada en el enfoque de ThickDB como la capa más interna, no se presenta en este diagrama, ya que es la capa donde se encuentra el propio RDBMS, donde están los objetos de la base como las tablas e índices y donde se ejecutan las sentencias `SELECT`, `INSERT`, `UPDATE` y `DELETE` ofrecidas por el dialecto específico de SQL del RDBMS. En la capa que llamamos *DL Code*, se encuentran los procedimientos y/o funciones donde se permite utilizar las sentencias SQL. Esta capa ofrece hacia la capa superior una interfaz para manipular las tablas de la base como si fueran un Tipo Abstracto de Datos o *Abstract Data type* (ADT), por sus siglas en inglés. Es interesante notar que esta capa se puede generar y mantener sincronizada con el esquema de la base de forma automática, y podría mantenerse un catálogo de las funciones existentes para ser utilizadas por la capa exterior. También podría ofrecer una interfaz estándar independiente del RDBMS si existiera un DBPL estándar, y el hecho de que tal lenguaje no exista en la actualidad no significa que no pueda desarrollarse en el futuro.

En la capa que llamamos *BL Code*, se encuentra la lógica del dominio, que utiliza las funcionalidades provistas por la capa anterior. Esta capa podría a su vez subdividirse en función de las necesidades, bien en capas más internas y capas más externas, o bien en segmentos para funcionalidades transversales, como reportes o sistemas de autenticación y autorización. La interfaz de esta capa hacia el exterior, ofrece las operaciones de alto nivel en formato de procedimientos con los tipos de datos específicos del RDBMS.

Las capas de *DL Code* y *BL Code*, juntas, se corresponderían con el anillo intermedio en el enfoque de ThickDB.

En la capa que llamamos *Interface Wrapper* se consumen las operaciones

de alto nivel y se vuelven a ofrecer hacia las capas superiores pero con las entradas y salidas en un formato estándar independiente tanto de los tipos de datos del RDBMS como de la interfaz, como por ejemplo *Extensible Markup Language* (XML) o JSON.

Finalmente, en la capas UI Code y *Service Interface* (SI) Code se incluyen dos opciones diferentes de formato de la salida, una diseñada para clientes finales a través de un *web browser*, y otra diseñada para *Web Services*. Vale mencionar que aquí "interfaz" no se refiere únicamente a los elementos de una interfaz gráfica, y esto permite tratar de una forma simétrica el uso de las funcionalidades de la capa de dominio tanto para un usuario humano que requiere una interfaz HTML como para otro sistema que invoca un *Web Service*. En la medida que no haya elementos de interfaz en ninguna de las capas subyacentes, el mantenimiento de la interfaz en esta capa debería resultar más fácil en la medida que quede circunscripto a esta. Debe notarse que la arquitectura es una propuesta teórica, y que pueden tener sentido variantes en función de las decisiones de implementación. Por ejemplo, si se decidiera adoptar algún producto como PostgREST [96] para simplificar la provisión automática de una API REST a partir de los *stored procedures* de la capa del *BL Code*, la capa del *Interface Wrapper* podría suprimirse.

A partir de este esquema general, queda la cuestión de dónde quedará ubicada la *engine* del IDE para desarrollar aplicaciones y de qué forma interactuará con cada una de las capas, por lo que es necesario diferenciar entre la aplicación IDE y las aplicaciones de usuario final.

En principio, el IDE podría ser una aplicación que siguiera la misma arquitectura, con sus propios objetos de base de datos, sus procedimientos de DML en la capa del *DL Code*, su lógica de dominio en la capa del *BL Code*, donde estará propiamente la *engine* y su *wrapper* de interfaz en las capas *Interface Wrapper* y *UI Code*. Un desarrollador de aplicaciones debería poder crear y mantener una aplicación de usuario final utilizando el IDE. Este IDE debería tener funcionalidades para poder crear y mantener las aplicaciones de usuario final: desde los objetos de base de datos, incluyendo una forma automática o semiautomática de mantener sincronizada una capa de procedimientos DML sobre los objetos de la base, pasando por una forma de mantener la capa de la lógica del dominio, y hasta las funcionalidades que permitan crear la interfaz de usuario e interactuar con la capa lógica del dominio.

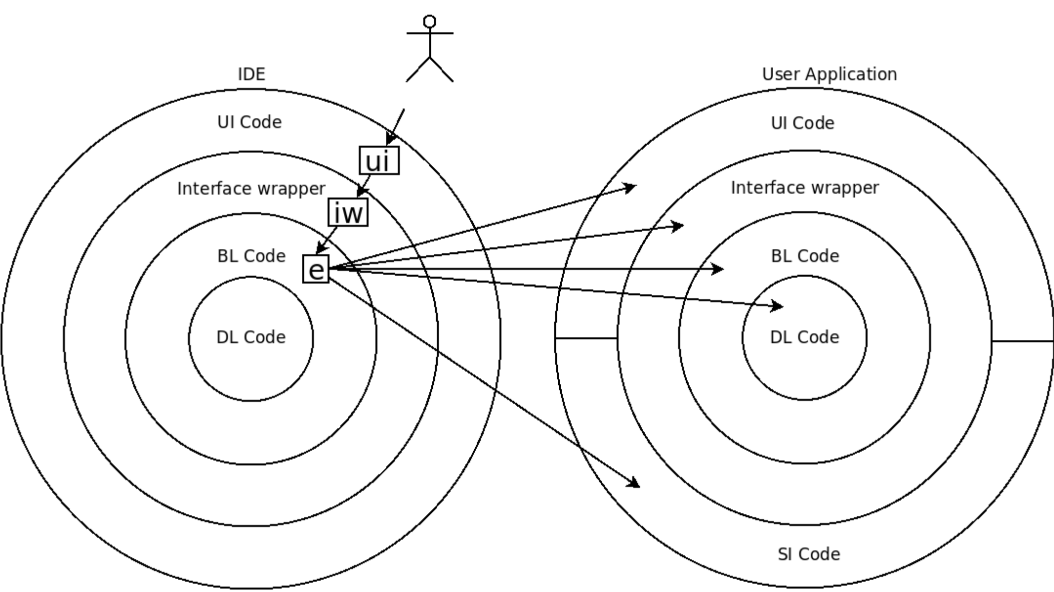


Figura 4.2: Interacción entre IDE y aplicación de usuario

Sin embargo, debe notarse que la *engine* del IDE debe conocer e interactuar con los objetos de la capa de interfaz de las aplicaciones de usuario, lo que representa una excepción a la regla inicial que exigía que "los procedimientos de cada capa solo tienen permitido conocer objetos e invocar procedimientos de la capa inmediatamente inferior". Esta excepción es insalvable, y acaso pueda justificarse notando que la interfaz de las aplicaciones es parte del dominio del negocio de un IDE. La Figura 4.2 presenta la interacción entre una aplicación de tipo IDE y una aplicación de usuario final. En esta figura, las capas se reemplazaron por anillos concéntricos para hacer más clara la necesidad de que la *engine* (parte de la capa *BL Code*) conozca y manipule objetos de todas las capas de la aplicación de usuario final.

La Figura 4.3 presenta un segundo acercamiento a la arquitectura que incluye la *engine*, excluida de la estructura en capas del resto de la arquitectura.

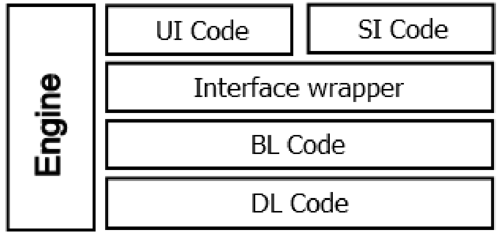


Figura 4.3: Segundo acercamiento a la arquitectura

En el siguiente capítulo se presenta una prueba de concepto, donde se desarrolló un prototipo tecnológico siguiendo los lineamientos presentados en este capítulo.

Capítulo 5

Prueba de concepto

> [...] the sciences use fundamentally the same method that common sense employs, the method of trial and error
>
> ---
>
> *All life is problem solving*
> *Karl R. Popper*

Este capítulo describe una prueba de concepto siguiendo los lineamientos del Capítulo 4 pero con un alcance limitado al desarrollo de las funcionalidades centrales que permitirían definir y ejecutar una aplicación. El desarrollo de una aplicación verdaderamente funcional se presenta en el Capítulo 6, y queda explícitamente fuera del alcance de este trabajo el desarrollo de un IDE.

Los componentes básicos de una tecnología *RDBMS-only* son:

1. Una aplicación desarrollada en el DBPL de un RDBMS, y
2. Un *web listener* que traduzca de HTTP al DBPL

La tecnología desarrollada en el DBPL de un RDBMS debería contener todas las responsabilidades de la generación estática y dinámica de sus páginas. Siguiendo los lineamientos del Capítulo 4 el código se dividió en esquemas de base de datos. El *web listener* simplemente debe encargarse de traducir los *HTTP Requests* a ejecuciones de los *stored procedures* en el RDBMS.

La prueba de concepto fue desarrollada con PostgreSQL como RDBMS y PL/pgSQL como DBPL, y un módulo de Apache como *web listener*, con capacidad de ejecutar PL/pgSQL, al que llamamos *mod_plpgsql*. Que el RDBMS no fuera Oracle era una cuestión de especial interés, ya que *a*

priori cabría preguntarse si la factibilidad de desarrollar un producto como APEX dependía de alguna carecterística inherente de este RDBMS. Se utilizó PostgreSQL por ser un producto *Free and Open Source Software* (FOSS) con un lenguaje procedural maduro, pero se podría haber elegido cualquier otro RDBMS. En la Sección 5.1 se presenta el módulo *mod_plpgsql*, que podría ser utilizado de forma independiente para desarrollar cualquier prototipo de aplicación. En la Sección 5.2 se discuten algunas particularidades sobre el manejo del estado de las sesiones. En la Sección 5.3 se presenta un prototipo de tecnología llamada *webpg* utilizando el módulo *mod_plpgsql* y siguiendo los lineamientos de la arquitectura *RDBMS-only* de la Figura 4.1. Este prototipo de tecnología se centra en el desarrollo de la *engine* que permite el desarrollo de aplicaciones de usuario final. En la Sección 5.4 se presentan algunas consideraciones relevantes sobre los *endpoints* para atender los *HTTP Requests* por los métodos *HTTP GET* y *HTTP POST*. Finalmente, en la Sección 5.5 se presentan algunas indicaciones sobre el proceso de desarrollo y testing de aplicaciones en el prototipo.

5.1. El módulo mod_plpgsql

En esta sección se presentan las generalidades del desarrollo del módulo *mod_plpgsql*. Su objetivo es servir como un *web listener*, y si bien la idea general es idéntica a la del módulo *mod_plsql* presentada en las secciones 3.3 y 3.4, se discuten aquí algunos detalles técnicos interesantes, y lecciones aprendidas durante la implementación. Vale aclarar que solamente se incluyen aquellos detalles técnicos que podrían aportar a la comprensión del funcionamiento interno del módulo.

Es importante aclarar que este módulo no contiene ningún tipo de lógica, y eso es lo que hace que esta propuesta sea cualitativamente diferente a cualquier arquitectura donde exista una capa lógica, como cuando se cuenta con un contenedor de aplicaciones como un Tomcat o incluso en los casos de lenguajes como PHP o Perl. En este sentido, conviene ver el módulo simplemente como un *web listener*, un traductor de HTTP al DBPL del RDBMS. La Figura 5.1 presenta esta idea.

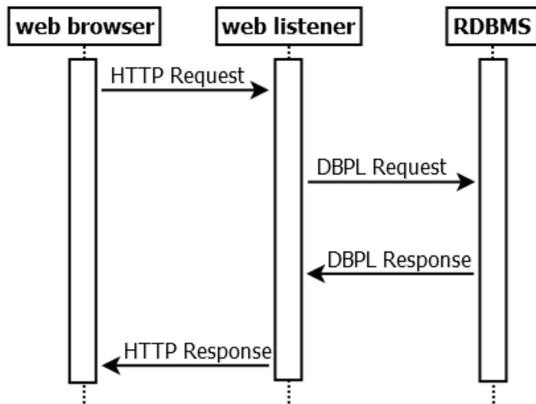


Figura 5.1: El módulo *mod_plpgsql* como un traductor HTTP/DBPL

Desde un punto de vista lógico, el módulo se encuentra entre el Apache y el RDBMS PostgreSQL. La Figura 5.2 presenta las interacciones del módulo: el *web server* Apache le delega *HTTP Requests* y él se encarga de ejecutar *stored procedures* en el RDBMS, recuperar una salida que será el código de una página HTML y enviarla como *HTTP Response* al cliente. Vale insistir en que esto es radicalmente diferente de lo que hace cualquier otra arquitectura que tenga su lógica de negocio fuera del RDBMS.

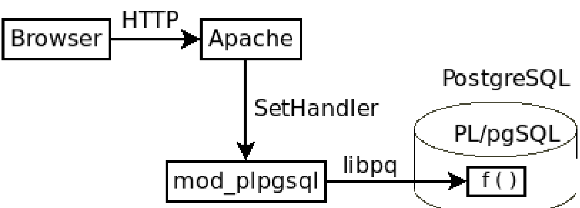


Figura 5.2: Interacciones del módulo *mod_plpgsql*

Se utilizó el paquete *apache2-dev* y mediante el utilitario `apxs` se creó un módulo en lenguaje C llamado *mod_plpgsql*. Una vez compilado, este módulo debe cargarse en un servidor web Apache mediante una configuración como la que se presenta en la Figura 5.3, y debe ser habilitado mediante un comando `a2enmod`. Vale mencionar que todo el módulo *mod_plpgsql* se implementó con menos de 600 líneas de código C.

```
$ cat /etc/apache2/mods-available/plpgsql.load
LoadFile /usr/lib/x86_64-linux-gnu/libpq.so
LoadModule plpgsql_module
           /usr/lib/apache2/modules/mod_plpgsql.so

$ cat /etc/apache2/mods-available/plpgsql.conf
<IfModule plpgsql_module>
   # Allow server  to handle PL/pgSQL requests
   <Location /plpgsql>
      SetHandler plpgsql
   </Location>
   LogLevel debug
</IfModule>
```

Figura 5.3: Configuración del módulo plpgsql

Una vez instalado y habilitado el módulo, cada vez que llegue un *HTTP Request* que incluya el prefijo `/plpgsql` en el recurso, Apache transferirá al módulo la responsabilidad de manejar el *HTTP Request*. En el código del módulo, una función `plpgsql_handler` recibe una estructura de tipo `request_rec` y debe encargarse de formar y devolver el *HTTP Response*. Una de las primeras validaciones que se realizan es que la URL incluya un nombre que se encuentre registrado en un archivo `plpgsql.conf`. Por ejemplo, en el caso que la URL comience con `http://host:port/plpgsql/demorep` el módulo recuperará, de una sección llamada `demorep` dentro del archivo de configuración, la información de la *Uniform Resource Identifier* (URI) de conexión para conectarse a una base PostgreSQL.

```
# Configuration for config demorep
# PGCONNECT_URI format is:
# postgresql://[user[:password]@][netloc][:port][/dbname]
# LOG_LEVEL is INFO or DEBUG
[demorep]
PGCONNECT_URI=postgresql://repusr:reppwd@dumbo:5432/rep
LOG_LEVEL=INFO
```

Figura 5.4: Configuración del plpgsql.conf

En la Figura 5.4 se muestra un ejemplo de sección en el archivo de configuración que indica, para la sección `demorep`, la URI de conexión a la base (en la variable `PGCONNECT_URI`) y un nivel de logueo (en la variable `LOG_LEVEL`). Debe advertirse que una vez compilado e instalado el módulo, puede configurarse para conectarse a múltiples bases de datos agregando secciones

en el archivo plpgsql.conf.

En el caso del *HTTP GET*, el módulo obtiene de la URL el nombre de la función que debe ejecutar y el parámetro que debe pasarle. Por ejemplo, si la URL es http://host:port/plpgsql/demorep?reports&report=employees, el módulo se conecta a la base mediante la URI asociada a la sección demorep que recupera del plpgsql.conf y ejecuta una función PL/pgSQL llamada reports, que debe existir previamente y debe aceptar exactamente un parámetro *varchar* de entrada y un parámetro *varchar* de salida. La Figura 5.5 muestra un ejemplo de la firma que debe tener una función PL/pgSQL para poder ser llamada desde el módulo *mod_plpgsql*.

```
1 CREATE FUNCTION reports(
2    p_repname         character varying,
3    OUT p_repcontent character varying
4 ) RETURNS character varying
```

Figura 5.5: Firma de una función PL/pgSQL llamada desde mod_plpgsql

La llamada que se realiza es un SELECT sobre la función pasándole el primer parámetro de la URL. En la Figura 5.6 se muestra el resultado de llamar a la función de la forma prevista desde una consola *psql*, y es la misma llamada que se realiza desde el módulo. La función debe devolver la página HTML que el módulo enviará en el *HTTP Response*, y toda la responsabilidad de interpretar el parámetro y construir la página que formará la respuesta se delega a la función PL/pgSQL.

```
1 rep# select * from reports('report=employees');
2 p_repcontent
3 ------------------------------------------------------------
4 <html><head><title>plpgsql reports</title></head><body>
5 ...
6 <a href="?reports&report=main">Return to the main page</a>
7 ...
```

Figura 5.6: Llamada a la función PL/pgSQL

Vale destacar que si bien la firma de la función prevé un único parámetro de tipo *varchar*, es posible que la función determine alguna convención sobre el formato, con algún caracter que delimite parámetros conceptuales. De esta forma, la función puede encargarse tanto de separar los parámetros

conceptuales del parámetro de entrada, como de construir las URLs de los *links* de la página de respuesta que permitirán la navegación por la aplicación. Así, es posible que las funciones PL/pgSQL implementadas en el RDBMS tengan una gran libertad para implementar aplicaciones sin modificar el módulo *mod_-plpgsql*.

Un ejemplo de convención sobre el formato podría ser definir que existirá una función llamada `g`, que será un *endpoint* para atender los *HTTP GET* y que recibirá un parámetro que será de la forma `app_id:page_id`, siendo `app_id` un número de aplicación y `page_id` un número de página. Se notará que esta convención fue tomada siguiendo el caso de APEX, al igual que la estructura básica de la jerarquía: aplicación → página → región → ítem. Así, puede lograrse que un *HTTP GET* a la URL `http://host:port/plpgsql/myapp?g&p=1:1` se conecte a la base de datos que determine la sección `myapp` del `plpgsql.conf` y ejecute un `SELECT` sobre la función `g`, pasándole como parámetro `"p=1:2"` que tendría la semántica de "aplicación 1, página 2". En la Figura 5.7 se presenta un ejemplo de lo que puede retornar esta función en la base, logrando la navegabilidad hacia otra página de la misma aplicación. La información para construir la página en cuestión estará dentro del esquema donde se define la aplicación, el que tiene una estructura similar al de la Figura 3.1 del Capítulo 3.

```
1  myappdb# select * from g('p=1:2');
2  p_repcontent
3  ------------------------------------------------------------
4  <html><head><title>App 1 - Page 2</title></head><body>
5  ...
6  <!-- region: 4 (links) -->
7  <a href="?g&p=1:3">Go to page 3</a>
8  ...
```

Figura 5.7: Determinación de parámetros por convención

5.2. Manejo de estado

Lo presentado hasta el momento es suficiente para lograr una aplicación sin estado. Sin embargo, es interesante notar que si se quiere mantener una sesión a través de un protocolo sin estado como HTTP, se debe recurrir a una *cookie*, y como el módulo debe ejecutar una función específica tanto para setear la

cookie en el *HTTP Header* como para recuperarla, debe extenderse la lógica del módulo en una de las formas siguientes:

1. Extender las funciones de la base de forma que tengan un parámetro adicional de entrada y un parámetro adicional de salida
2. Extender la lógica del módulo de forma que sea capaz de modificar el parámetro de entrada para incorporar el valor de la *cookie* y analizar el parámetro de salida para recuperar la información de la *cookie*

En este trabajo se optó por la segunda alternativa para no aumentar la complejidad de la firma de las funciones pero fundamentalmente porque la gestión de *cookies* está conceptualmente relacionada al protocolo HTTP y es una funcionalidad lo suficientemente estándar como para mantener al *web listener* como un componente fundamentalmente tecnológico. Cuando llega un *HTTP Request* sin sesión, se debe interpretar como una sesión nueva, debe crearse una sesión y se debe agregar en la respuesta una indicación para que el módulo *mod_plpgsql* agregue el *header Set-Cookie* en el *HTTP Response*, además de que todos los *links* o formularios de la página se deben generar incluyendo el parámetro de la sesión. Como medida de seguridad rudimentaria, el módulo *mod_plpgsql* intercepta el valor de la *cookie* enviada por el *browser*, el que que se interpreta como el número de sesión, y lo agrega al parámetro, de forma que si el parámetro original era `"p=1:2"` y el valor de la *cookie* era `12345678`, el parámetro se transforma en `"p=1:2:12345678"`. De esta forma, es posible verificar que el valor de la sesión en el *HTTP Request* coincide con el valor de la sesión en la *cookie*[1]. Al RDBMS se le delega la responsabilidad de gestionar las sesiones, incluyendo su expiración.

Vale mencionar que las sesiones no siempre implican autenticación, es decir, tener identificado al usuario conectado. Hay casos donde es útil o necesario mantener una sesión aún cuando no sabemos la identidad del usuario. Por ejemplo, en una aplicación de tipo *e-commerce*, es común que un usuario, sin necesidad de registrarse, vaya realizando compras y manteniendo el estado de un "carrito de compras". En el momento que el usuario decide terminar la compra y pagar, se procede a la identificación y pago, pero se necesitó una

[1] Por esto mismo, es interesante notar que podría prescindirse de pasar el valor de la sesión por la URL. En este trabajo no se realizó la supresión por implicar un esfuerzo de desarrollo que no aportaba en lo esencial. Es curioso que APEX hasta la versión 21.1 sigue pasando el valor de la sesión en la URL, cuando no parece haber ningún motivo que justifique la necesidad

cookie durante todo el proceso previo, al menos a partir del momento en que el usuario agregó el primer ítem al carrito. En cualquier caso, es necesario definir por convención de qué forma las funciones PL/pgSQL le indicarán al módulo que debe agregar el *header Set-Cookie*, y también por convención de qué forma el módulo recuperará los valores de la *cookie* y los incluirá en las llamadas a las funciones PL/pgSQL. Todo esto se trata con más detalle en las secciones 5.3 y 5.4.

5.3. El prototipo de aplicación webpg

En esta sección se presentan las generalidades de un prototipo de aplicación llamado *webpg*, que utiliza el módulo *mod_plpgsql* y funciones PL/pgSQL en una base de datos PostgreSQL.

En la Figura 5.8 se presenta lo fundamental de una vista de procesos de la arquitectura, mediante la relación entre las principales funciones del prototipo.

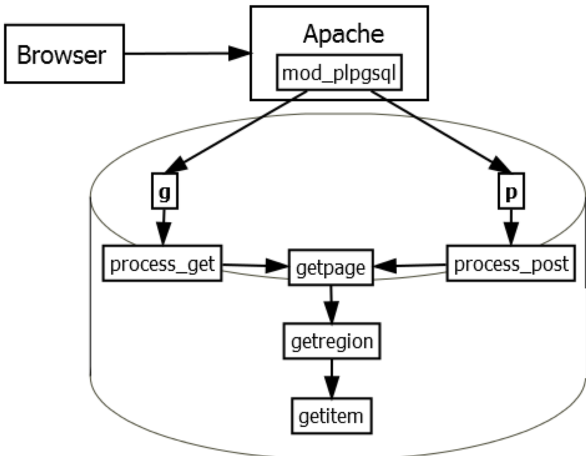


Figura 5.8: Principales funciones del prototipo webpg

Una decisión importante de diseño es la sintaxis del parámetro de entrada de los *HTTP Requests*, tanto en el caso de los *GET* como de los *POST*. Para este prototipo se tomaron las siguientes decisiones de diseño:

- Para los *HTTP Requests* con método *GET*, las URLs serán de la forma `http://host:port/plpgsql/webpg?g&p=app:[page]:[session]`
- Para los *HTTP Requests* con método *POST*, las URLs serán de la forma `http://host:port/plpgsql/webpg?p`, y los parámetros se pasarán en formato JSON {`app_id:app,page_id:page,session_id:session,...`}

La sintaxis es arbitraria, pero debe estar definida *a priori*, de manera que cuando se desarrolle una aplicación se puedan implementar las navegaciones ya sea en forma de *links* o de *HTML Forms*.

En el caso del *GET* tanto la página como la sesión son opcionales. Si no se especifica una página, la aplicación asume que se solicita una página por defecto de la aplicación, y si no se especifica una sesión, la aplicación asume que se trata del primer *HTTP Request* de un usuario y procede a crearle una sesión.

En el caso del *POST*, se debe especificar la página y la sesión debe existir. Típicamente habrá más parámetros que los indicados arriba, ya que si no se requiere el pasaje de parámetros, la navegación se podría resolver con un *GET*.

Como los parámetros de la URL en el caso del *GET* están fijados *a priori*, no es posible pasar parámetros arbitrarios. En el caso que una navegación necesite pasar parámetros, como sería el caso de navegar de una página de búsqueda de libros a una página con el detalle de un libro pasando su código, se debería implementar con un *POST*. En otras palabras, en lugar de un *link*, se debe implementar con un *HTML Form* que tenga parámetros (típicamente ocultos) `app_id`, `page_id`, `session_id` y `book_id`.

En la Figura 5.9 se presenta un ejemplo del código HTML necesario para implementar esta navegación en el prototipo, seguido de un *link* a la página principal. Para este ejemplo, se supone que existe una página con id = 1 que es la principal, y una página con id = 9 que presenta el detalle de un libro.

```
1  <form action="webpg/p" method="post">
2    <input type="hidden" name="app_id" value="1">
3    <input type="hidden" name="page_id" value="9">
4    <input type="hidden" name="session_id" value="10000012">
5    <input type="hidden" name="book_id" value="2459">
6    <input type="IMAGE" name="thumb_2459.gif"
7           src="data:image/gif;base64,/9j/4AAQ...2Q=="
8           width="100" height="150">
9  </form>
10
11 <a href="webpg?g&p=1:1:10000012">
12   <img id="home.gif" src="data:image/gif;base64,R01...AAOw==">
13 </a>
```

Figura 5.9: Ejemplo de código de navegación por GET y POST

Otro aspecto interesante de observar en la Figura 5.9 es la forma en que se genera el código para las imágenes. En lugar de referenciar un recurso de

filesystem y delegar al *web server* la responsabilidad de entregar las imágenes, se incluye la representación en *Base64* de las mismas en el código de la página, siguiendo el *data URL scheme* [97]. De esta forma, la entrega de las imágenes también queda dentro de las responsabilidades del RDBMS. Esta alternativa implica que el tamaño de una página con imágenes sea mayor, pero también que no se generarán *subrequests* para la descarga de las imágenes. El impacto de esta alternativa en el desempeño se discutirá oportunamente en la Sección 7.6.

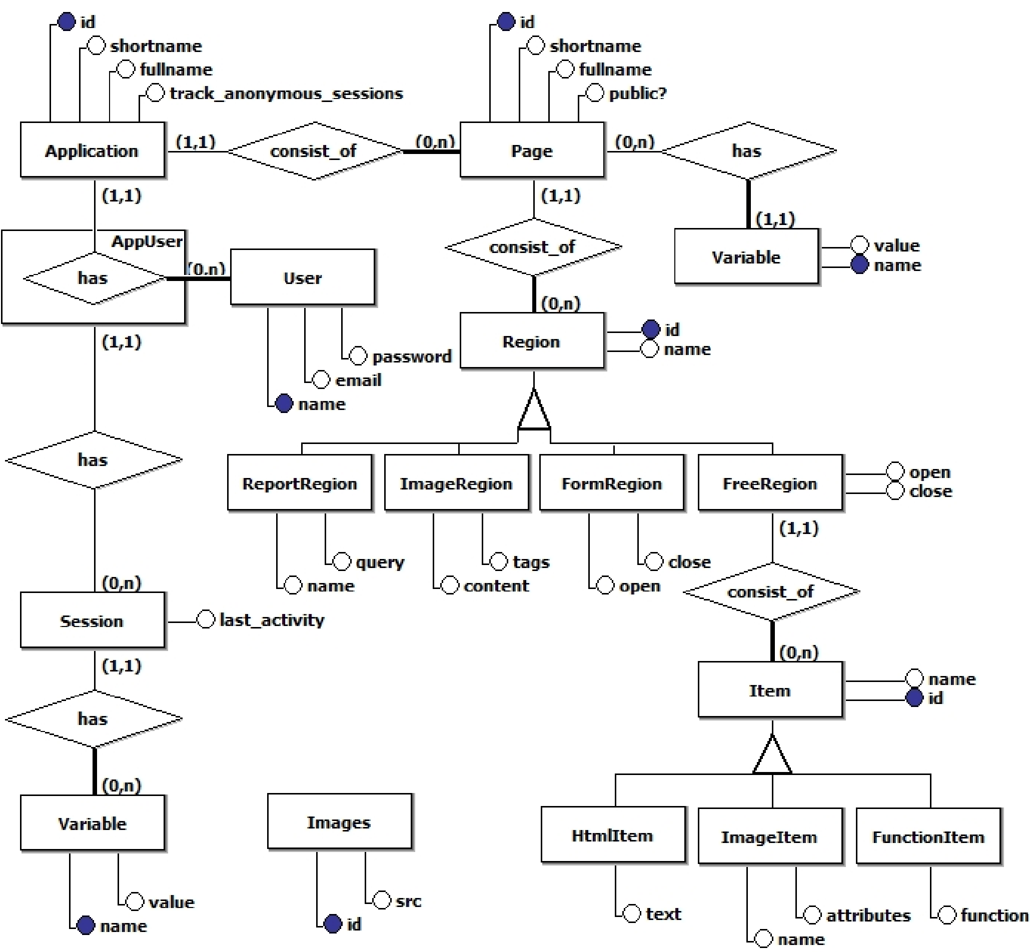


Figura 5.10: Modelo Entidad Relación del prototipo webpg

En la Figura 5.10 se presenta una vista lógica de la arquitectura del prototipo. Es importante apreciar la jerarquía fundamental aplicación → página → región → ítem, ya que una aplicación, en último análisis, será un conjunto de tuplas que poblará las tablas que se deriven de este MER. La relevancia de las demás entidades quedará de manifiesto cuando se discuta la

vista de procesos de la arquitectura. Vale mencionar que lo que aquí se presenta es el modelo original del prototipo, que *a posteriori* evolucionó a partir de los problemas que presentaba el desarrollo de una aplicación funcional.

Con un modelo como el de la Figura 5.10, la parte medular del prototipo consiste en el desarrollo de una función[2] `g` similar a la de la Figura 5.7 para atender los *HTTP Requests* con método *GET*; y de una función `p`, para atender los *HTTP Requests* con método *POST*. En cualquiera de los dos casos, debe notarse que más allá de la diferencia tecnológica, el problema fundamental es obtener una página de una aplicación. Por este motivo, como se puede ver en la Figura 5.8, se desarrolló una función genérica `getpage`, que pudiera servir tanto para el caso del *GET* como del *POST*, así como funciones intermedias `process_get` y `process_post` para resolver problemas particulares de los métodos *GET* y *POST* respectivamente.

La Tabla 5.1 presenta en muy alto nivel las responsabilidades de las funciones de la Figura 5.8.

Función	Responsabilidades
g	Validar el parámetro de la URL y generar un JSON para pasarle a la función process_get
p	Validar el parámetro como JSON, gestionar el caso especial de la página 0 (por convención, login) y llamar a la función process_post
process_-get	Validar que la aplicación existe, que la página existe dentro de la aplicación, manejar la sesión, llamar a la función getpage y sustituir variables de sesión e imágenes
process_-post	Validar que la aplicación existe, que la sesión es válida, insertar o actualizar variables de sesión, llamar a la función getpage y sustituir variables de sesión e imágenes
getpage	Gestionar las variables de página y generar el código de una página iterando en sus regiones por medio de getregion
getregion	Generar el código de una región, eventualmente iterando en sus ítems por medio de getitem
getitem	Generar el código de un ítem

Tabla 5.1: Responsabilidades de las principales funciones del prototipo webpg

[2]Se utiliza el término función, ya que en PL/pgSQL los objetos de tipo *function* son los *stored procedures* que retornan un valor, lo cual es necesario en estos casos

A continuación se describe una vista de desarrollo de las funciones de la Tabla 5.1. La Sección 5.4 tratará algunos detalles técnicos de la invocación de las funciones `g` y `p` por parte del módulo *mod_plpgsql*.

La Figura 5.11 presenta el código simplificado de la función `g`, omitiendo el manejo de excepciones. Ésta valida que el parámetro `p_param` sea algo como `"p=1:1:0:0"` e invoca algo como `"process_get ('{appid:1, pageid:1, sessionid:0, cookiesid:0}')"`.

```
1  create or replace function g(p_param varchar,
2                               out p_pagecontent varchar)
3    returns varchar language plpgsql as $$
4  declare
5    v_prefix    varchar;
6    v_paramlist varchar;
7    v_keys      varchar;
8    v_lista     varchar;
9    v_jsonparam jsonb;
10 begin
11   v_prefix=substr(p_param, 1, 2);
12   if (v_prefix <> 'p=') then
13     p_pagecontent = 'Malformed URL';
14     return;
15   end if;
16   v_paramlist = substr(p_param, 3, length(p_param)-2);
17   select array(select name from urlparams order by position)
18   into v_keys;
19   v_lista = regexp_split_to_array(v_paramlist, E'\\:+');
20   v_jsonparam = jsonb_object(v_keys::text[], v_lista::text[]);
21   select process_get_and_getpage(v_jsonparam)
22   into p_pagecontent;
23   return;
24 end;
25 $$;
```

Figura 5.11: Código simplificado de la función g

En la Tabla 5.1, la función `process_get` tenía dentro de sus responsabilidades "manejar la sesión". Como esta responsabilidad es una de las más interesantes, se presenta su implementación en la Figura 5.12. Vale destacar que el prototipo permite dos comportamientos para las aplicaciones respecto de las sesiones, en función del atributo `track_anonymous_sessions` de la entidad *Application*, como se puede observar en la Figura 5.10. Uno de estos comportamientos implica crear una sesión únicamente cuando se pasa por una página de login, y el otro implica crearle una nueva sesión a todo usuario que no disponga de una sesión válida.

```
if (is_valid_session(v_appid, v_sesid, v_cookiesid)) then
  update sessions set last_activity = current_timestamp
  where session = v_sesid and application = v_appid;
else
  if (track_anonymous_sessions(v_appid)) then
    v_sesid = create_session(v_appid, 'ANONYMOUS')::int;
  else
    if (is_public_page(v_pageid)) then
      v_sessionid = 0;
    else
      select getformat('AUTH_ERR') into p_pagecontent;
      return;
    end if;
  end if;
end if;
```

Figura 5.12: Manejo de la sesión en la función process_get

Con lo presentado hasta el momento es posible desarrollar una aplicación muy básica, pero este prototipo tuvo que evolucionar para poder soportar el desarrollo de una aplicación un poco más compleja como la que se presenta en el Capítulo 6. Sin embargo, como esas líneas de evolución surgieron a partir de algunas necesidades concretas, se posterga su presentación hasta ese capítulo donde se cuenta con el contexto de esas necesidades.

5.4. Los endpoints para HTTP GET y HTTP POST

Le llamaremos *"endpoints"* a las funciones PL/pgSQL desarrolladas para ser ejecutadas desde el *web listener*, ya que pueden verse como un servicio que el RDBMS ofrece. Los nombres de estos *endpoints* y sus parámetros se pueden elegir de forma arbitraria, pero cualquier cambio en sus firmas implica que el *web listener* debe adaptarse, por lo cual una elección que permita abarcar casos de uso universales agrega valor. Por este motivo, la elección de los *endpoints* genéricos `g` y `p`, que permiten atender cualquier *HTTP Request* con método *GET* o *POST* que llegue al *web listener* parece una idea razonable, pero quizás no sea suficiente si se desea hacer evolucionar el prototipo para adaptarse a un uso real. Direcciones de evolución que podrían requerir el desarrollo de nuevos *endpoints*, podrían ser el soporte de asincronía (por ejemplo mediante AJAX) y el soporte de *Representational State Transfer* (REST) *Web Services*.

En este último caso, por ejemplo, se requeriría un *endpoint* con soporte a todos los verbos de una API REST. Aunque el análisis de estas posibilidades ha quedado fuera de alcance, se presentan los detalles técnicos del desarrollo de los *endpoints* `g` y `p` con la intención de facilitar un eventual intento de desarrollar uno nuevo.

En la Figura 5.13 se presenta un extracto del comienzo de la función `plpgsql_handler`, que conceptualmente se denomina *content handler* y es la función que recibe de Apache la responsabilidad de gestionar los *HTTP Requests* cuyo recurso comience con `"plpgsql"`. Esta función recibe una referencia a una estructura de tipo `request_rec` que contiene toda la información del *HTTP Request*, y en primer lugar valida que los métodos sean *GET* o *POST*.

```
1  static int plpgsql_handler(request_rec *r) {
2
3    if (strcmp(r->handler, "plpgsql")) {
4      return DECLINED;
5    }
6    r->content_type = (char *)"text/html";
7
8    if (!strcmp(r->method,"GET") && !strcmp(r->method,"POST")) {
9      ap_rprintf(r, "Method %s not supported", r->method);
10     return OK;
11   }
```

Figura 5.13: Extracto del inicio del código del content handler

En la Figura 5.14 se presenta un extracto de la función `plpgsql_handler` donde se obtiene el valor de la *cookie* del *HTTP Request* y se gestiona el método *GET*. Una función auxiliar obtiene el *config* dentro del `plpgsql.conf`, así como el nombre de la función, el parámetro y un indicador de modo de *debug* (que por defecto es “NO” y solo se establece en “YES” si la URL incluye al final un segundo parámetro `"&debug"`). Para una URL de la forma `http://bookshop.com/plpgsql/ecommerce?g&p=1:1:0` se tendría que el *config* es `ecommerce`, la función es `g`, el parámetro es `p=1:1:0` y debug es `NO`. Otra función auxiliar verifica la existencia del *config* en el `plpgsql.conf`. El parámetro se modifica para agregar la sesión obtenida de la *cookie* para determinar si ya existía una sesión, y se genera en la variable `function_call` la función tal como será ejecutada en el RDBMS, que en el caso del ejemplo y suponiendo que no existiera una sesión sería `"g('p=1:1:0:0')"`.

Una función auxiliar se encarga de realizar la ejecución de la consulta utlizando la biblioteca *libpq*, que en nuestro caso sería `"select p_pagecontent from g('p=1:1:0:0')"`, y se inspecciona el resultado para verificar si incluye una indicación de que debe setearse un identificador de sesión en la *cookie*. Finalmente, se devuelve el *HTTP Response* al cliente.

```
1  char * cookie = (char*)apr_table_get(r->headers_in, "Cookie");
2  if (!strcmp(r->method, "GET")) {
3  ...
4  int uriok = parsear_uri_get(r->path_info, r->args, &config,
5                              &function, &parameter, &debug);
6  ...
7  if ( uriok == 0 ) {
8    configok = verificar_config(strdup(config), &name,
9                                &pgconnect_uri, &log_level);
10   if ( configok == 0 ) {
11     char * session = getSessionValueFromCookie(r);
12     parameter = apr_pstrcat(r->pool, parameter, ":",
13                             session, NULL);
14     char * function_call = (char *) malloc(strlen(function)
15                                   +strlen(parameter)+5);
16     sprintf(function_call, "%s(\'%s\')", function, parameter);
17     char * response = (char *) malloc(MAXPAGESIZE);
18     response = ejecutar(pgconnect_uri, function_call);
19     if (startsWith(response, "<!--SetCookieSession:")) {
20       session = subString(response, 21, 8, session);
21       apr_table_set(r->err_headers_out, "Set-Cookie",
22                 apr_psprintf(r->pool, "session=%s", session));
23     }
24     ...
25     ap_rputs(response, r);
```

Figura 5.14: Extracto de código del content handler con la gestión del GET

Debe notarse que la indicación de setear un valor para la *cookie* es una convención que debe acordarse entre el módulo y la implementación en el RDBMS. En este caso, la convención es que la respuesta comience con un comentario HTML de la forma `<!--SetCookieSession:NNNNNNNN-->`. La responsabilidad de definir en qué casos debe hacerlo y con qué identificadores se delega al RDBMS, mientras que el módulo se limita a pasar la sesión de la URL, y la de la *cookie* en caso de existir.

En la Figura 5.15 se presenta un extracto de la función `plpgsql_handler` donde se gestiona el método *POST*. Si bien la idea general es la misma que en el caso del *GET*, la diferencia fundamental es que no pueden obtenerse parámetros de la URL, que siguiendo el ejemplo anterior sería de

la forma `http://bookshop.com/plpgsql/ecommerce?p`. Aquí los parámetros se deben obtener del *payload* del *POST*, y como no es posible convenir un número de parámetros *a priori*, se recuperan todos los parámetros y se forma un único *string* en formato JSON. Siguiendo con el ejemplo, la función ejecutaría en el RDBMS algo como: `"select p_pagecontent from p('{app_id:1, page_id:9, session_id:10000003, book_id:3174}')"`.

```
1  if (!strcmp(r->method, "POST")) {
2    int uriok = parsear_uri_post(r->path_info, &config,
3                                 &function);
4    if ( uriok == 0 ) {
5      configok = verificar_config(strdup(config), &name,
6                                  &pgconnect_uri, &log_level);
7      if ( configok == 0 ) {
8        char * session = getSessionValueFromCookie(r);
9        keyValuePair *formData = readPost(r);
10       char * parameter = "{";
11       ...
12       parameter = apr_pstrcat(r->pool, parameter, "}", NULL);
13       char * function_call = (char *) malloc(strlen(function)
14                                      +strlen(parameter)+5);
15       sprintf(function_call, "%s(\'%s\')",function,parameter);
16       char * response = (char *) malloc(MAXPAGESIZE);
17       response = ejecutar(pgconnect_uri, function_call);
18       if (startsWith(response, "<!--SetCookieSession:")) {
19         session = subString(response, 21, 8, session);
20         apr_table_set(r->err_headers_out, "Set-Cookie",
21                       apr_psprintf(r->pool, "session=%s",
22                                    session));
23       }
24       ap_rputs(response, r);
```

Figura 5.15: Extracto del código del content handler con la gestión del POST

Por simplicidad, la función `ejecutar` que es la que realiza el `SELECT` en el RDBMS, realiza una nueva conexión en cada ejecución. Para evitar el costo de la creación de las conexiones, puede utilizarse una estrategia de *pool* de conexiones, ya sea desarrollándolo en el módulo o incorporando a la solución un componente que realice este trabajo, como PgBouncer [98] o pgpool [99].

5.5. Proceso de desarrollo y testing

Es necesario aclarar qué se entiende en este contexto por “desarrollar” una aplicación dentro del prototipo, cuando no existe un IDE, ya que éste quedó

explícitamente fuera de alcance. El desarrollo de una aplicación consiste en poblar las tablas que se derivan del MER de la Figura 5.10, es decir que se reduce a escribir sentencias `INSERT` o `COPY` sobre las tablas `applications`, `pages`, `regions`, `items`, etc. Este proceso manual demostró ser engorroso, y llevó a una dinámica determinada por la prueba y el error. Ocasionalmente, además, se descubría que las funcionalidades del prototipo no eran suficientes para resolver alguna necesidad, y se debía proceder a modificar las funciones y eventualmente hasta las tablas de la *engine* del prototipo, para luego volver a intentar satisfacer la necesidad concreta en el desarrollo de la aplicación. Por este motivo, en este primer prototipo acotado, las funciones `getregion` y `getitem` acceden directamente a tablas que en el caso de una aplicación real tendrían los datos del dominio. Vale decir que esto viola la prescripción de la arquitectura propuesta en la Figura 4.1 del Capítulo 4, y se posterga la solución de este incumplimiento hasta el Capítulo 6 debido a que el foco en el desarrollo de una aplicación funcional ofrece el escenario ideal para trabajar en las capas propuestas.

Otra responsabilidad digna de ser mencionada de las funciones `process_get` y `process_post` es la sustitución de las variables de sesión y de las imágenes. En la Figura 5.16 se presenta una sentencia `INSERT` que determinará el contenido de la región 19 de la página 2 en la aplicación 1, y como se observa es un *link* a otra página como forma de resolver la navegación. Como es posible que sea necesario mantener la sesión, el código estático debe incluir una variable que pueda ser sustituida en tiempo de ejecución por la sesión del usuario. Se utilizó la convención de que el *string* `"###SESSION###"` sería esta variable, y se asignó a las mencionadas funciones de la *engine* la responsabilidad de realizar la sustitución.

```
1 insert into free_regions(app_id, page_id, region_id,
2                          region_open, region_close)
3 values (1, 2, 19,
4 '<a href="webpg?g&p=1:1:###SESSION###">Go to home</a>', '');
```

Figura 5.16: Variables de sesión en el código de las aplicaciones

Algo similar ocurre con las imágenes, que se mantienen en la tabla `images` con su nombre y su representación *Base64*. En la Figura 5.17 se presenta una sentencia `INSERT` que determinará el contenido de la región 22 de la página 6 en la aplicación 1, e incluye el nombre de una imagen. Se utilizó la convención

de que un *string* que satisfaga la expresión regular `"###IMAGE:([A-Za-z._-]*)###"` sería considerado una variable de imagen, y las funciones lo sustituirán por un *tag img* adecuado que contenga la imagen en su representación *Base64* como puede verse en la Figura 5.9.

```
1 insert into image_regions(application_id, page_id, region_id,
2                           region_content, image_tags)
3 values (1, 6, 22,
4 '###IMAGE:logo.gif###', 'BORDER="0" WIDTH="288" HEIGHT="67"');
```

Figura 5.17: Imágenes en el código de las aplicaciones

El nombre de las imágenes puede ser cualquier *string* que satisfaga la expresión regular, y no tiene por qué tener la forma usual de un nombre de archivo en *filesystem* con un punto y una extensión. Estas imágenes, en todo el proceso de uso de la aplicación, nunca llegan a estar como un archivo en *filesystem* ni del lado del servidor ni del lado del cliente.

El proceso de testing, al igual que el de desarrollo, demostró ser engorroso. A priori, podría pensarse que una tecnología de aplicaciones web que se reduce a la ejecución de *stored procedures* en un RDBMS sería fácil de testear, ya que todos los flujos se pueden reproducir en el contexto del RDBMS y es relativamente fácil automatizar estos tests. Si bien lo anterior es cierto, ocurre que para identificar la causa de los errores es necesario poder realizar el seguimiento del flujo de código a través de los diferentes *stored procedures*, y esto implica seguir un algoritmo de *backtracking* que puede tener varios niveles. Esta estrategia *top-down* se impuso al comienzo del trabajo más bien debido a una falta de estrategia de testing, y el análisis de esta complejidad derivó en dos estrategias diferentes:

- Utilizar un enfoque *bottom-up* con casos de prueba documentados para todos los *stored procedures*
- Implementar un mecanismo de debug, que pudiera prenderse y apagarse mediante un parámetro que se propague en las sucesivas llamadas

La prueba de concepto presentada en este capítulo demuestra la viabilidad del desarrollo de una aplicación web siguiendo la propuesta de la arquitectura *RDBMS-only*. Con lo presentado hasta aquí tenemos un prototipo del módulo *mod_plpgsql* que funciona, podemos obtener páginas desde el RDBMS, podemos gestionar el estado de las sesiones mediante cookies y tenemos un prototipo

de *engine* con una forma estándar de obtener páginas organizadas a partir de un MER especialmente diseñado para este objetivo y que permite procesar *HTTP Requests* por métodos *GET* y *POST*. A partir de estos rudimentos, el Capítulo 6 presenta una iteración incremental orientada al desarrollo de una pequeña aplicación.

Capítulo 6

Experimentos

> [...] la teoría campea en el trabajo experimental, desde que se establecen los planes iniciales hasta que se dan los últimos toques en el laboratorio.
>
> ---
>
> *La lógica de la investigación científica*
> *Karl R. Popper*

Se presentan en este capítulo algunos resultados de la experimentación con el prototipo *webpg*. Por un lado, se utilizó *webpg* durante el desarrollo de una aplicación simple de *e-commerce* extendiendo la prueba de concepto del Capítulo 5. Por otro lado, se realizan evaluaciones del desempeño de la aplicación desarrollada.

El desempeño de las aplicaciones web es un tema de gran interés, y existen discusiones acerca de la mejora en la eficiencia del cómputo cuando este puede moverse de un servidor de aplicaciones al RDBMS y viceversa [100]. Como en la arquitectura *RDBMS-only* no existe esta opción de mover cómputo del RDBMS a un servidor de aplicaciones, una pregunta válida es si existirá una penalización de desempeño y en qué medida. Por este motivo, fue de especial interés que la aplicación desarrollada con *webpg* sirviera específicamente al objetivo de medir el desempeño.

Se consideraron las alternativas de diseñar un pequeño sistema nuevo y de adoptar un pequeño sistema ya diseñado. Se optó por esta segunda alternativa,

y se encontraron varias ventajas en la adopción de un sistema especificado por el *Transaction Processing Performance Council* (TPC), una organización dedicada a crear y mantener *benchmarks* de desempeño. El sistema elegido fue el *TPC Web Commerce Benchmark*, también conocido como *TPC Benchmark W* (TPC-W), un *benchmark* específico para sistemas de *e-commerce* que especifica todos los detalles relevantes de una aplicación [101, 102].

Además de existir una especificación, el TPC-W cuenta con una implementación realizada con *Java Servlets* por la Universidad de Wisconsin [103, 104]. Existen al menos dos buenas razones para realizar una implementación alternativa en *webpg* de esta aplicación, fundamentadas en la existencia de una implementación previa. La primera es que podemos verificar si la aplicación implementada en *webpg* logra interfaces y comportamientos idénticos, y de lograrlo podríamos afirmar que con una arquitectura *RDBMS-only* es posible implementar una aplicación típica de *e-commerce*. La segunda es que podemos comparar ambas implementaciones según diversas características de calidad, pero especialmente desempeño, ya que el TPC-W prescribe métricas de desempeño por medio de las cuales se pueden comparar dos sistemas.

En la Sección 6.1 se presenta el TPC-W, así como las limitaciones que se fijaron al alcance con el objetivo de mantener el experimento dentro de unos límites de complejidad razonables para el presente trabajo. La Sección 6.2 presenta los primeros pasos del desarrollo siguiendo el paradigma de la arquitectura *RDBMS-only* y algunas particularidades de la aplicación TPCW. Como el desarrollo en una tecnología *RDBMS-only* comienza con el modelo de datos y sigue hacia arriba en la arquitectura de capas de la Figura 4.3 del Capítulo 4, se presentan algunas consideraciones del desarrollo de las capas del *DL Code* y *BL Code*. La Sección 6.3 presenta una posible implementación de una capa de *DL Code* mediante la cual las tablas pueden ser accedidas a través de una interfaz estándar. La Sección 6.4 presenta un ejemplo de implementación de una capa de *BL Code* con las operaciones necesarias para resolver los casos de uso. La Sección 6.5 presenta una discusión sobre la capa de *Interface Wrapper* cuya implementación quedó fuera de alcance, y finalmente la Sección 6.6 presenta un ejemplo de implementación de una capa de *UI Code*.

6.1. El TPC Benchmark W

En esta sección se describen los aspectos fundamentales de la especificación del *System Under Test* (SUT) del TPC-W, en cuanto a comportamiento, modelo de datos e interfaz gráfica.

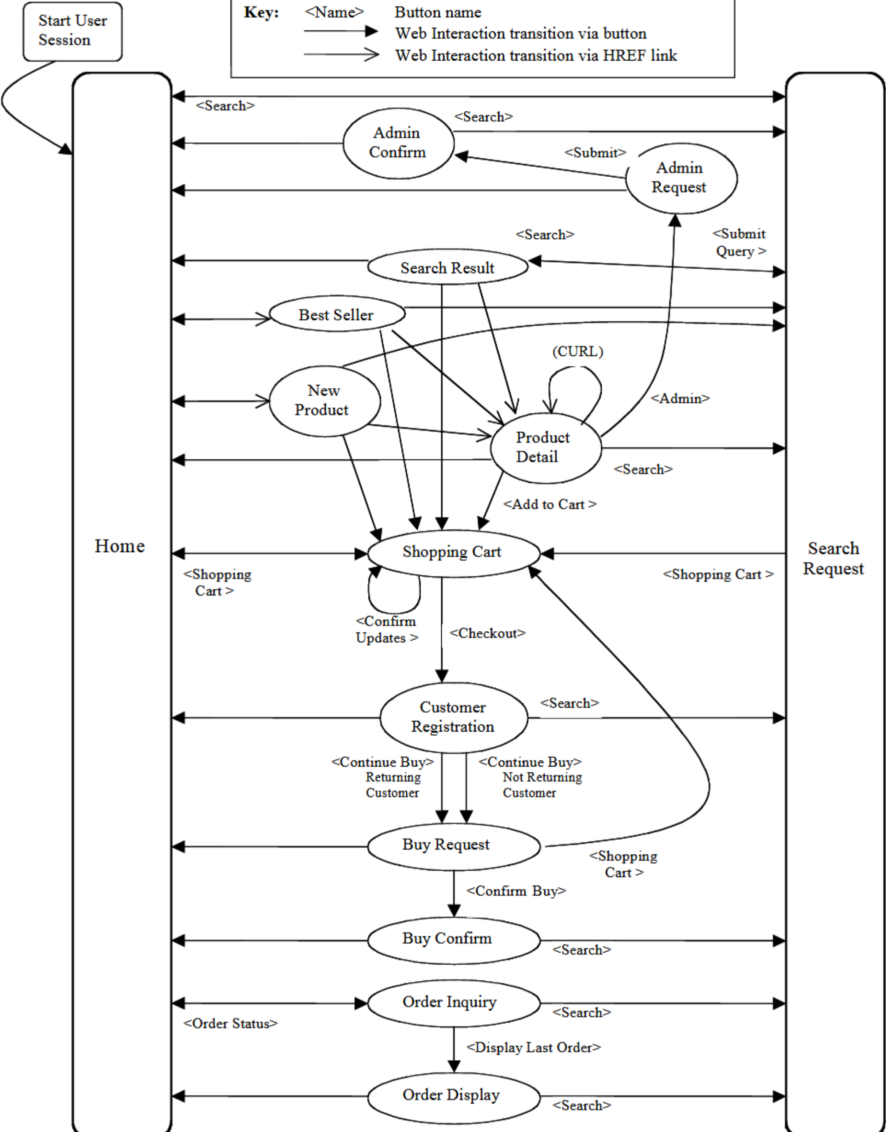


Figura 6.1: Páginas y navegación en el *TPC Benchmark W*. Fuente: *A Methodology for Workload Characterization of E-commerce Sites* [105]

El SUT pretende modelar un pequeño sistema de *e-commerce* donde se pueden buscar y comprar libros. Para ilustrar el comportamiento, la Figura 6.1 describe las 14 páginas y posibles navegaciones, mediante un grafo de transición de estados denominado *Customer Behavior Model Graph* (CBMG) [105].

Un caso de uso típico sería comenzar en la página principal *"Home"*, ir a un link de género, como Biografías, dentro de la sección *"Best Sellers"*, elegir un libro dentro de la lista retornada e ir su detalle *"Product Detail"*, agregarlo al carrito de compras llegando a *"Shopping Cart"*, finalizar la compra y registrarse en *"Customer Registration"*, continuar la compra y ver el detalle de la compra a realizar en *"Buy Request"*, y finalmente confirmar la compra terminando en *"Buy Confirm"*, desde donde puede volverse a la página principal *"Home"* o puede realizarse una búsqueda en *"Search Request"*.

El modelo de datos también se prescribe en el *benchmark*. La Figura 6.2 presenta las 8 tablas que debe tener el esquema mínimo. Se omiten el resto de las especificaciones sobre el esquema como tipos de datos, o claves primarias y foráneas; y también se omite cualquier especificación sobre la instancia como la cantidad de tuplas de cada tabla.

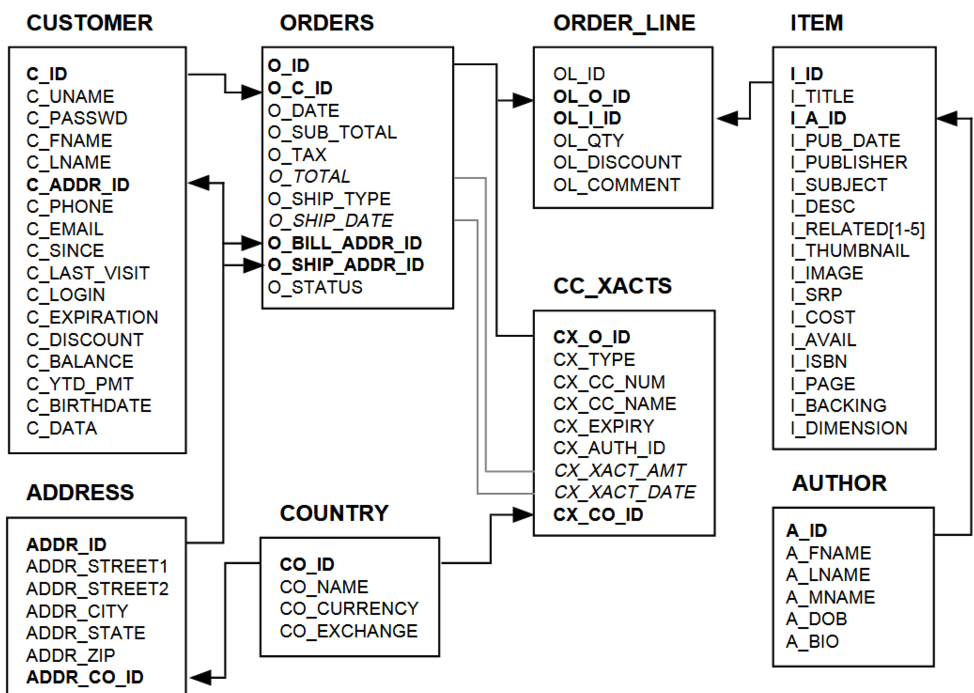


Figura 6.2: Modelo lógico del *TPC Benchmark W*. Fuente: *TPC Benchmark W Specification* [101]

La interfaz también se describe en el *benchmark*, y en la Figura 6.3 se muestra un ejemplo de la *Home Page*. Si bien no es un requerimiento que la interfaz se vea exactamente igual, se describen varias restricciones sobre el HTML de cada una de las páginas, como cantidad mínima de caracteres, existencia del logo, imágenes para los botones o las cinco miniaturas de

cubiertas de libros incluyendo el algoritmo de su selección. El *benchmark* también ofrece código HTML de ejemplo para cada una de las páginas.

Figura 6.3: Interfaz de la *Home Page* del *TPC Benchmark W*. Fuente: captura de pantalla de la implementación de Wisconsin

Para mantener acotado el alcance de este trabajo, no se implementaron todas las navegaciones posibles sino algunas, elegidas de forma que se traten de cubrir los casos más comunes (según la propia especificación del *benchmark*) o más interesantes (según criterios técnicos de implementación).

6.2. La aplicación TPCW

El primer paso del desarrollo en una aplicación *RDBMS-only* es el diseño conceptual en un MER que permita satisfacer los requerimientos funcionales, pero como en este caso la aplicación a implementar es el SUT que se describe en el *benchmark* TPC-W, no se realizó la etapa del diseño conceptual, y el diseño lógico es el que viene dado por la Figura 6.2. Se presentarán algunos detalles del desarrollo de esta aplicación mostrando su definición en las tablas del modelo de la Figura 5.10 de la Sección 5.3, así como ejemplos del HTML generado de forma dinámica.

La Figura 6.4 presenta las 14 páginas de la aplicación TPCW que implementan las páginas especificadas por el TPC-W en la Figura 6.1 (más una página de *login*). Cada página de una aplicación se corresponde con una tupla de la tabla `pages`.

```
1  tpcw=> select page_id, shortname from pages
2         where application_id = 1 order by page_id;
3   page_id |          shortname
4  ---------+----------------------------
5         0 | TPC-W Login
6         1 | TPC-W Home
7         2 | TPC-W Shopping Cart
8         3 | TPC-W Order Inquiry
9         4 | TPC-W Order Display
10        5 | TPC-W Search Request
11        6 | TPC-W Search Results
12        7 | TPC-W New Products
13        8 | TPC-W Best Sellers
14        9 | TPC-W Product Detail
15       10 | TPC-W Customer Registration
16       11 | TPC-W Buy Request
17       12 | TPC-W Buy Confirm
18       13 | TPC-W Admin Request
19       14 | TPC-W Admin Confirm
20 (15 rows)
```

Figura 6.4: Páginas de la aplicación TPCW

El desarrollo comenzó por la definición de las páginas más estáticas, como la *"Home"*. Pese a ser una página casi estática, su desarrollo presentó dos desafíos diferentes. El primero, la imposibilidad de implementar como *tags* `<a>` los *links* que se ven en las secciones *"What's New"* y *"Best Sellers"* de la Figura 6.3, ya que en *webpg* no es posible incluir

parámetros arbitrarios en la URL, ya que en la Sección 5.3 se definió que para los *HTTP Requests* con método *GET*, las URLs serán de la forma `http://host:port/plpgsql/webpg?g&p=app:[page]:[session]`. El segundo, la obtención de los cinco libros para mostrar sus miniaturas, según un procedimiento aleatorio específico, definido en la sección *promotional processing* del *benchmark*.

En general, el enfrentamiento con estos problemas experimentales fue de mucha ayuda para identificar problemas en la implementación del Capítulo 5, y es así que los experimentos que se presentan en esta sección permitieron aumentar y mejorar el prototipo implementado en el Capítulo 5. Por ejemplo, el desafío de replicar el *promotional processing* llevó a la necesidad de implementar las variables de página e ítems de tipo función, lo que se explica más adelante en esta misma sección.

La Figura 6.5 presenta el código generado para resolver la navegación pasando parámetros, y corresponde al *link* con texto *"ARTS"* que se ve en la sección *"What's New"* de la Figura 6.3. El código JavaScript solo es necesario por motivos estéticos, para que visualmente se vea como un *link*. Por todo el resto de esta sección los ejemplos tendrán siempre una aplicación con identificador 1.

```
1 <form id="wn_subj_01" action="tpcw/p" method="post">
2   <input type="hidden" name="app_id" value="1">
3   <input type="hidden" name="page_id" value="7">
4   <input type="hidden" name="session_id" value="10000000">
5   <input type="hidden" name="subject_id" value="ARTS">
6   <a href="javascript:{}"
7      onclick="document.getElementById('wn_subj_01').submit();
8               return false;">ARTS</a>
9 </form>
```

Figura 6.5: Pasaje de parámetros en *HTML Forms*

El *HTML Form* de la Figura 6.5 genera un *POST* al *endpoint* `p` solicitando la página 7 y pasándole como parámetro `"subject_id=ARTS"` (además de los parámetros propios de la navegación en *webpg*). Cuando esta solicitud se procesa, debe retornar una lista de los últimos títulos del género *"ARTS"*. La responsabilidad de generar esa lista se delegará al *BL Code* y se presentará en la Sección 6.4.

El desafío de replicar el *promotional processing* para obtener los cinco libros cuyas miniaturas se ven en la Figura 6.3 implicó la utilización de las variables

de página e ítems de tipo función. El problema se reduce a la generación dinámica de un *HTML Form* que tenga como *hidden inputs* `app_id`, `page_id`, `session_id` y `book_id`, así como una imagen en miniatura de la cubierta del libro que funcione como botón de *submit*. La Figura 6.6 presenta la definición de estas variables de página, que referencian a una función `get_random_book()` de la capa del *BL Code*.

```
1   tpcw=> select * from page_variables
2          where application_id = 1 and page_id = 1;
3    application_id | page_id | var_name |      var_value
4   ----------------+---------+----------+-------------------
5                 1 |       1 | book1_id | get_random_book()
6                 1 |       1 | book2_id | get_random_book()
7                 1 |       1 | book3_id | get_random_book()
8                 1 |       1 | book4_id | get_random_book()
9                 1 |       1 | book5_id | get_random_book()
10  (5 rows)
```

Figura 6.6: Variables de página para la obtención aleatoria de libros

Las variables de página se procesan al comienzo de la función `getpage`, y sus valores se registran en las variables de sesión, desde donde se recuperan después. La Figura 6.7 presenta las variables de sesión donde se mantienen los identificadores de los cinco libros para la sesión 1000000 en la página 1.

```
1   tpcw=> select * from session_variables
2          where session = 10000000 and page = 1;
3    session   | page | var_name | var_value
4   ----------+------+----------+-----------
5    10000000 |    1 | book1_id | 9242
6    10000000 |    1 | book2_id | 2569
7    10000000 |    1 | book3_id | 2435
8    10000000 |    1 | book4_id | 1008
9    10000000 |    1 | book5_id | 8210
10  (5 rows)
```

Figura 6.7: Variables de sesión

La definición estática de la página debe hacer uso de funciones que recuperen los valores de estas variables de sesión para poder generar de forma dinámica el *HTML Form*. La Figura 6.8 presenta parte de la definición de la página 1, donde se especifican los ítems de función. Algunas de las funciones generan información estática, como el caso de la función `get_hidden` del *UI*

Code que simplemente retorna el código HTML para un *hidden input*. Otras generan información dinámica, como el caso de la función `get_session_variable` que permite obtener el valor de la variable de sesión `book1_id`.

```
1  tpcw=> select item_id, item_name, item_type from items
2         where application_id = 1 and page_id = 1
3         and region_id = 2 and item_id between 2 and 6;
4   item_id |    item_name     | item_type
5  ---------+-----------------+-----------
6         2 | RandomBook 1 h1 | function
7         3 | RandomBook 1 h2 | function
8         4 | RandomBook 1 h3 | function
9         5 | RandomBook 1 h4 | function
10        6 | RandomBook 1    | function
11 (5 rows)
12
13 tpcw=> select item_id, function from function_items
14        where application_id = 1 and page_id = 1
15        and region_id = 2 and item_id between 2 and 6;
16  item_id |                   function
17 ---------+------------------------------------------------------
18        2 | get_hidden('app_id', '1')
19        3 | get_hidden('page_id', '9')
20        4 | get_hidden('session_id', '###SESSION###')
21        5 | get_hidden('book_id',
22 get_session_variable('###SESSION###', 1, 'book1_id'))
23        6 | get_inputimg_from_name(get_thumb_name(
24 get_session_variable('###SESSION###', 1, 'book1_id')),
25 'WIDTH="100" HEIGHT="150"')
26 (5 rows)
```

Figura 6.8: Items de función

Las funciones pueden anidarse, y pueden ser cualquier función *built-in* del RDBMS o definida por el usuario en cualquiera de las capas, ya que su valor se evalúa en tiempo de ejecución mediante SQL dinámico. En la Figura 6.8 puede observarse que el ítem 5 genera un *hidden input* con el `book_id` cuyo valor se obtiene de una variable de sesión, y el ítem 6 genera un elemento de interfaz de tipo *inputimg* con la representación en *Base64* de la imagen a partir del nombre de la misma, la que a su vez fue obtenida a partir de una variable de sesión con el identificador de un libro.

La Figura 6.9 presenta el código del *HTML Form* generado a partir de los ítems de función de la Figura 6.8.

```
<form action="tpcw/p" method="post">
  <!-- function item 2 -->
  <input type="hidden" name="app_id" value="1">
  <!-- function item 3 -->
  <input type="hidden" name="page_id" value="9">
  <!-- function item 4 -->
  <input type="hidden" name="session_id" value="10000000">
  <!-- function item 5 -->
  <input class="" type="hidden" name="book_id" value="3072">
  <!-- function item 6 -->
  <input type="IMAGE" name="thumb_3072.gif"
         src="data:image/gif;base64,/9j/4AAQ...TUMsqZZVw0Ms//Z"
         width="100" height="150">
</form>
```

Figura 6.9: *HTML Form* de un *random book*

En las siguientes secciones se discute cómo se reparten las responsabilidades entre las capas de la Figura 4.1 para generar las partes dinámicas de las páginas, como el extracto de la Figura 6.9, a partir de la información estática, como la del extracto de la Figura 6.8.

6.3. Capa de DL Code

Se desarrolló un conjunto de *stored procedures* que hacen uso intensivo de metaconsultas en el catálogo de PostgreSQL y generan funciones de `INSERT`, `UPDATE` y `DELETE` en todas las tablas de un esquema. En la Figura 6.10 se presenta una jerarquía con las principales funciones para generar este *DL Code*. La función de más alto nivel, `generate_dl_for_schema()`, recibe como parámetro el nombre de un esquema y retorna un *script* completo para generar o regenerar el *DL Code* del esquema.

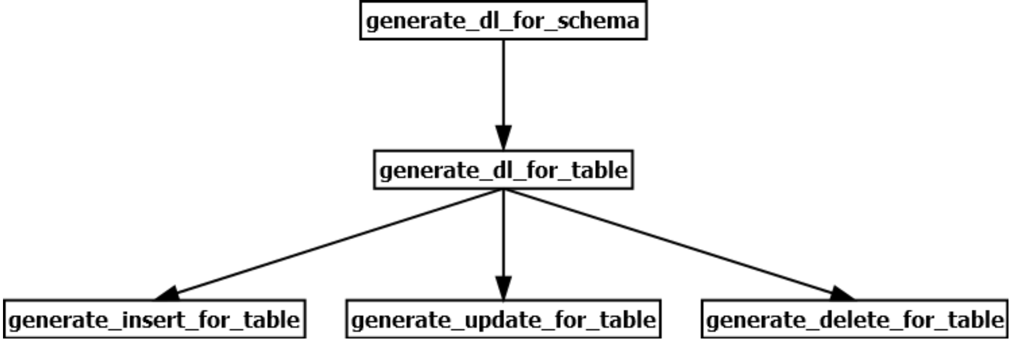


Figura 6.10: Funciones para generación automática del *DL Code*

Una vez creada y poblada una base, se puede generar la capa de *DL Code* como se ve en la Figura 6.11. El script `generate_dl.sql` no se presenta por razones de espacio, contiene el código de los *stored procedures* que se ven en la Figura 6.10 (y otros que no se ven) y que se encargan de recorrer el catálogo haciendo todo lo necesario para la generación automática de la capa de *DL Code*.

```
1 psql -d tpcw -U tpcw -f generate_dl.sql
2
3 psql -d tpcw -U tpcw -qtA \
4      -c "select generate_dl_for_schema('public')" \
5      > tcp_dl.sql
6
7 psql -d tpcw -U tpcw -f tcp_dl.sql
```

Figura 6.11: Generación del *DL Code* para la base tpcw

Las funciones del *DL Code* pueden utilizarse para lograr cierta independencia de las tablas, y su uso podría justificarse al menos por dos motivos. Primero, la capa del *DL Code* podría residir en un esquema separado del de las tablas, como una forma de ocultar la implementación y garantizar el acceso a los datos a través de una interfaz estandarizada. Segundo, esta capa de funciones generadas de manera uniforme y automática permite implementar cualquier particularidad requerida en las funciones sin modificar la interfaz.

La modificación directa del código generado de forma automática y la regeneración automática cuando se modifica el esquema son dos posibilidades excluyentes. En cualquier caso, podría implementarse una forma de establecer tabla por tabla, cuáles se mantendrán sincronizadas automáticamente y cuáles quedarán bajo control del usuario. Esta implementación no se realizó, y queda abierta la discusión sobre las diferentes maneras de hacer evolucionar la capa del *DL Code*.

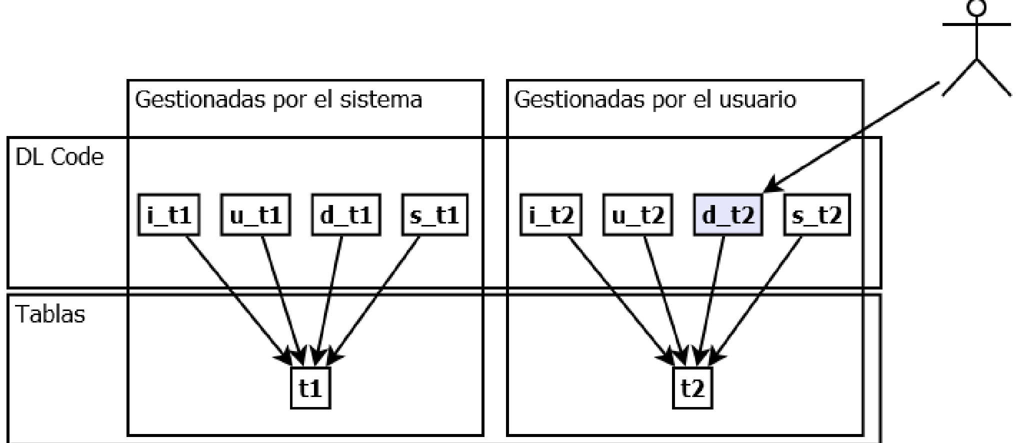


Figura 6.12: Funciones del *DL Code*

A modo de ejemplo, la Figura 6.12 pretende ilustrar algunas de las posibilidades de evolución del *DL Code*. La función `generate_dl_for_table(t1)` genera las funciones `i_t1`, `u_t1`, `d_t1` y `s_t1` (`INSERT`, `UPDATE`, `DELETE` y `SELECT` en la tabla `t1`, respectivamente). Si se pretende que esta tabla `t1` se mantenga gestionada por el sistema, el código de las funciones `i_t1`, `u_t1`, `d_t1` y `s_t1` no debería modificarse directamente. Cualquier modificación que se realice de forma indirecta, a través de las funciones de la Figura 6.10, podría quedar disponible de forma homogénea para todas las tablas que estén gestionadas por el sistema. Esto podría ser útil para mantener soluciones de auditoría de datos o de sentencias, de seguridad basada en condiciones de las tuplas, o de enmascaramiento, redacción o cifrado de datos. De esta forma, las capacidades de un RDBMS se pueden llegar a aumentar mediante la incorporación de funcionalidades en esta capa. También podría darse la flexibilidad de mantener algunas tablas gestionadas por el usuario, para poder modificar sus funciones de acceso de forma directa, y que queden excluidas del `generate_dl_for_schema()`.

Cabe destacar que si bien se podrían generar de forma automática funciones de consulta para no ejecutar sentencias `SELECT` directamente sobre las tablas, esto no se implementó. Por un lado, debido a que la forma simple de implementar esto es con funciones que retornen una tabla, y sobre éstas también sería necesario acceder con sentencias `SELECT`, con lo cual no se obtiene ningún beneficio salvo el del control del acceso, que igualmente puede obtenerse por la vía del control de los privilegios. Por otro lado, los beneficios más importantes de implementar este tipo de funciones consisten en estandarizar

funcionalidades comunes como el paginado, pero debido a la complejidad de esta implementación, se decidió mantener fuera de alcance. En una eventual implementación de un IDE que pretendiera abarcar todas las etapas del desarrollo, sería fundamental integrar el mantenimiento de las tablas y la automatización del *DL Code* con funciones estandarizadas.

6.4. Capa de BL Code

El *BL Code* es la capa de la lógica del negocio de la aplicación, y donde típicamente se invertirá más esfuerzo en una aplicación compleja. En la Figura 6.13 se presentan un par de ejemplos de funciones simples del *BL Code* de la aplicación TPCW, donde se puede apreciar que las entidades que se manejan tienen semántica dentro del dominio del negocio, como *book* y *subject*. Estas dos funciones del *BL Code* se utilizan para la generación dinámica de la página principal de la Figura 6.3: la función `get_random_book()` es la que se utiliza en las variables de página como puede verse en la Figura 6.6, y la función `get_subjects()` se utiliza, a través de otra función del *UI Code*, para generar la lista de géneros literarios que aparecen como *links* en las secciones *"What's New"* y *"Best Sellers"* de la página principal.

```
1  -- get random book
2  create or replace function get_random_book()
3  returns varchar as $$
4  declare
5    v_book varchar(15);
6  begin
7    select floor(random()*9999+1)::varchar into v_book;
8    return v_book;
9  end;
10 $$ language plpgsql;
11
12 -- get subjects
13 create or replace function get_subjects()
14 returns table(subject varchar) as $$
15 begin
16   return query
17     select distinct i_subject from item order by i_subject;
18 end;
19 $$ language plpgsql;
```

Figura 6.13: Ejemplos de funciones del *BL Code*

La imagen de la Figura 6.14 muestra cómo se utilizan algunas funciones

del *BL Code* en la definición de la página *"Product Detail"* (página 9) de la aplicación TPCW. A esta página se puede llegar desde varias otras, pero siempre mediante un POST que incluye el `book_id`, el cual se mantiene en la tabla de variables de sesión, y puede ser recuperado mediante la función `get_session_variable()`.

```
tpcw=> select region_id as r, item_id as i, function
       from function_items
       where application_id = 1 and page_id = 9;
 r | i  |                         function
---+----+---------------------------------------------------
 2 |  2 | get_book_title(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 2 |  4 | get_book_author(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 2 |  6 | get_book_subject(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 3 |  1 | get_book_cover(get_session_variable
                 ('###SESSION###', 9, 'book_id'), 'ALIGN...')
 3 |  3 | get_book_description(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 4 |  2 | get_book_srp(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 4 |  4 | get_book_cost(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 4 |  6 | get_book_save(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 5 |  2 | get_book_backing(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 5 |  4 | get_book_page(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 5 |  6 | get_book_publisher(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 5 |  8 | get_book_pub_date(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 5 | 10 | get_book_avail(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 5 | 12 | get_book_dimensions(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
 5 | 14 | get_book_isbn(get_session_variable
                             ('###SESSION###', 9, 'book_id'))
(15 rows)
```

Figura 6.14: Utilización de funciones del *BL Code*

Es de esperar que en aplicaciones complejas el foco del desarrollo se encuentre en esta capa, incluyendo mecanismos avanzados de manejo de excepciones.

6.5. Capa de Interface Wrapper

El desarrollo de la capa del *Interface Wrapper* quedó fuera de alcance debido a su complejidad y a los pocos beneficios que esta capa ofrecía en el particular contexto limitado de este trabajo. Sin embargo, es posible comentar algunos aspectos de esta complejidad así como las razones por las que se consideró de poco beneficio.

La complejidad de esta capa reside en la necesidad de diseñar un formato de información estructurada, que modele de forma completa todos los elementos necesarios para generar tanto una interfaz gráfica como una respuesta de *Web Service*. Este formato podría estar basado en XML o JSON, pero debería definirse la estructura que modelara todos los elementos posibles en cualquier tipo de respuesta. La Figura 6.15 ofrece un ejemplo de esta dificultad mediante un primer acercamiento al formato de la respuesta del *Interface Wrapper* basado en JSON. En la línea resaltada puede apreciarse código HTML, que no debería existir en esta capa. En su lugar, debería existir una estructura de información con todos los elementos necesarios para generar la interfaz gráfica.

```
{
  "page": {
    "session": "1000000",
    "title": "TPC-W Home",
    "pagevars": [
      {
        "name": "book1_id",
        "value": "9276"
      }
    ],
    "regions": [
      {
        "id": "1",
        "open": "<!-- Region 1 open code -->",
        "items": [
          {
            "type": "html",
            "code": "<H1 ALIGN=\"center\">TPC Web Commerce Benchmark (TPC-W)</H1><H2 ALIGN=\"center\">"
          }
        ],
        "close": "<!-- Region 1 close code -->"
      }
    ]
  }
}
```

Figura 6.15: Acercamiento al formato del *Interface Wrapper*

No es esperable que este formato del *Interface Wrapper* tenga la misma capacidad de información que una interfaz en HTML. Este formato interno podría no incluir información relativa a la interfaz, como colores o alineación de un texto, pero sí debería tener la riqueza conceptual suficiente para permitir la

generación de una interfaz compleja. A partir de este formato se deberían poder generar interfaces diferentes en función de un tema que podría ser elegido por el usuario y/o en función del dispositivo. Esto permitiría agregar capacidades de *Web Accessibility*, o *Responsive Web Design*.

La ventaja de la codificación como JSON, es que permitiría de forma casi directa convertir la respuesta del *Interface Wrapper* en la respuesta de un *Web Service* REST. Es importante notar que el hecho de que este formato pueda ser transformado en una respuesta de un *Web Service*, no significa que esta no pueda ser utilizada por otro sistema para generar una interfaz gráfica. Es posible que dependiendo del uso que pretenda dársele a una respuesta no todos los elementos sean requeridos, y por motivos de desempeño podría ser conveniente tener una forma de limitar la inclusión de algunos elementos, por ejemplo las imágenes de los botones. Con esta idea en mente, la sustitución de los nombres de las imágenes por su contenido es una responsabilidad que debería delegarse a la capa del *UI Code*.

Debido a las restricciones de alcance de este trabajo, donde la única forma de interactuar con la aplicación sería a través de una interfaz gráfica única, el desarrollo de la capa del *Interface Wrapper* no ofrecía beneficios directos. Quedan abiertas las discusiones sobre los parámetros a incluir en las llamadas a las funciones de esta capa, así como el formato de la respuesta, y las capacidades a integrar.

6.6. Capa de UI Code

La capa del *UI Code* se presenta en un estado que parece artesanal, debido a que el desarrollo de un IDE quedó fuera de alcance. Típicamente, este IDE tendría la responsabilidad de mantener un catálogo de elementos de interfaz y permitiría generar el *UI Code* maquetando las páginas y vinculando este *UI Code* con el *Interface Wrapper*.

Al haber quedado fuera de alcance tanto el IDE como el *Interface Wrapper*, la capa del *UI Code* se limita a generar el código HTML necesario para la interfaz de la aplicación. Vale mencionar que para soportar AJAX o la entrega de JavaScript y CSS por medio de *subrequests* se deberían desarrollar endpoints y funcionalidades específicas para esos fines, en el estado actual pueden incluirse JavaScript y CSS en forma *inline*, es decir, dentro del HTML como en el caso de la Figura 6.5. La Figura 6.16 presenta la definición de la

región 3 de la página *"Home"* de la aplicación TPCW (Correspondiente a las secciones *"What's New"* y *"Best Sellers"* que pueden verse en la Figura 6.3), y especialmente los ítems de tipo *function* que llaman a funciones `get_subject_table()` del *UI Code*. El objetivo de esta función es generar una tabla con un *link* por cada género literario, que permita navegar a la página *"New Products"* (página 7) o *"Best Sellers"* (página 8) para ver una lista de los títulos más nuevos o más vendidos, respectivamente, del género.

```
1  tpcw=> select item_id, item_name, item_type from items
2         where application_id = 1 and page_id = 1
3         and region_id = 3;
4   item_id |   item_name   | item_type
5  ---------+---------------+-----------
6         1 | twotabsopen   | html
7         2 | whatsnewimage | image
8         3 | whatsnewtable | function
9         4 | middletable   | html
10        5 | bestsellimage | image
11        6 | bestselltable | function
12        7 | twotabsclose  | html
13 (7 rows)
14
15 tpcw=> select item_id, function from function_items
16        where application_id = 1 and page_id = 1
17        and region_id = 3;
18  item_id |                     function
19 ---------+-----------------------------------------------------
20        3 | get_subject_table('wn_subject_', '1', '7',
21                    '###SESSION###', '300', '0', '#c0c0c0')
22        6 | get_subject_table('bs_subject_', '1', '8',
23                    '###SESSION###', '300', '0', '#c0c0c0')
24 (2 rows)
```

Figura 6.16: Definición de las secciones *"What's New"* y *"Best Sellers"*

La función `get_subject_table()`, en la capa del *UI Code*, genera dinámicamente el contenido de una tabla HTML de dos columnas con los *links* para navegar a una página que se pasa como parámetro a la función. La Figura 6.17 presenta el código de la función, donde se puede apreciar que tiene la responsabilidad de generar los *tags* HTML de la tabla, iterando en el resultado de la función `get_subjects()` del *BL Code* (que puede verse en la Figura 6.13) para generar un *link* por cada género. La responsabilidad de generar un *link* se delega a la función `get_subject_link()`, y esta separación de responsabilidades puede verse como un intento de aplicar atributos deseables

de la modularidad como alta cohesión y bajo acoplamiento, que no solo tienen sentido en el paradigma de la Programación Orientada a Objetos sino también en la Programación Imperativa, ya que permiten mantener una alta cohesión de intereses en el código de cada función y lograr cambios más localizados [106].

```
create or replace function get_subject_table(
  p_formid_prefix varchar,
  p_app_id        varchar,
  p_page_id       varchar,
  p_session_id    varchar,
  p_width         varchar,
  p_border        varchar,
  p_color         varchar
) returns varchar as $$
declare
  v_counter       smallint;
  v_subject       varchar(30);
  v_form_id       varchar(30);
  v_subject_link  varchar(500);
  v_table         varchar(20000);
begin
  v_counter = 0;
  v_table = '<table width="'||p_width||'" border="'||p_border
          ||'"bgcolor="'||p_color||'">';
  for v_subject in select subject from get_subjects()
  loop
    v_counter = v_counter+1;
    v_form_id = p_formid_prefix||v_counter;
    v_subject_link = get_subject_link(v_form_id, p_app_id,
                                      p_page_id, p_session_id,
                                      v_subject, v_subject);
    if (v_counter%2=0) then
      v_table = v_table||'<td><center>'||v_subject_link
              ||'</center></td></tr>';
    else
      v_table = v_table||'<tr><td><center>'||v_subject_link
              ||'</center></td>';
    end if;
  end loop;
  return v_table||'</table>';
end;
$$ language plpgsql;
```

Figura 6.17: Código de la función `get_subject_table()`

La Figura 6.18 presenta el código de la función `get_subject_link()`, que se limita a generar un *HTML Form* que se vea y se comporte como un *link*.

```
create or replace function get_subject_link(
  p_form_id    varchar,
  p_app_id     varchar,
  p_page_id    varchar,
  p_session_id varchar,
  p_subject_id varchar,
  p_link       varchar
) returns varchar as $$
declare
  v_link varchar(1000);
begin
  v_link = '<form id="'||p_form_id
         ||'" action="tpcw/p" method="post">';
  v_link = v_link||get_hidden('app_id', p_app_id);
  v_link = v_link||get_hidden('page_id', p_page_id);
  v_link = v_link||get_hidden('session_id', p_session_id);
  v_link = v_link||get_hidden('subject_id', p_subject_id);
  v_link = v_link||'<a href="javascript:{}"
                    onclick="document.getElementById('''
                 ||p_form_id||''').submit(); return false;">'
                 ||p_link||'</a>';
  v_link = v_link||'</form>';
  return v_link;
end;
$$ language plpgsql;
```

Figura 6.18: Código de la función `get_subject_link`

Un ejemplo de código HTML de salida de esta función es el presentado en la Figura 6.5.

A modo de resumen, vale la pena describir de forma detallada, y a la luz de las funciones vistas, lo que sucede cuando alguien accede a la página *"Home"* de esta aplicación y sigue el *link* *"ARTS"* de la sección *"What's New"*. En primer lugar, al cargarse la página *"Home"* se reutiliza o se genera un número de sesión dependiendo de que el usuario ya tuviera o no una sesión válida, y se setean las variables `book1_id` a `book5_id` en la tabla `session_variables` (como se ve en la Figura 6.7) mediante la función del *BL Code* `get_random_book()` (que se puede ver en la Figura 6.13), según lo especifican las variables de página (como se puede ver en la Figura 6.6). Como se describe en la Tabla 5.1, después de procesar las variables de página y cuando ya tiene las nuevas variables de sesión disponibles, la función `getpage()` se encarga de generar el contenido de la página, región por región a través de la función `getregion()`, e ítem por ítem a través de la función `getitem()`. Para generar la región 2 donde se encuentran las 5 cubiertas de los libros aleatorios, se utilizan ítems

de función (como se puede ver en la Figura 6.8). Cada una de las cubiertas de los libros aleatorios funciona conceptualmente como un botón de *submit* de un *HTML Form* que permite navegar a la página *“Product Detail”* (página 9), pasando como parámetro el `book_id`, donde se puede ver el detalle del libro. Para generar la región 3 donde se encuentran las secciones *“What’s New”* y *“Best Sellers”* se utilizan ítems de función que invocan a la función `get_subject_table()` (como se puede ver en la Figura 6.16). La función `get_subject_table()` (cuyo código se puede ver en la Figura 6.17), utiliza a la función `get_subjects()` (cuyo código se puede ver en la Figura 6.13) para iterar en los géneros de los libros, y a la función `get_subject_link()` (cuyo código se puede ver en la Figura 6.18) para generar cada uno de los *HTML Forms* que se ven como *links*. El código HTML del *link* *“ARTS”* de la sección *“What’s New”* se puede ver en la Figura 6.5. Cuando se sigue este *link*, se genera un *POST* al *endpoint* `p` pasándole como parámetros `app_id=1`, `page_id=7` y `subject_id=ARTS` (además del valor de `session_id`). Como respuesta, la página *“New Products”* (página 7) muestra una lista de los 50 títulos más nuevos del género *“ARTS”*, según los datos de la tabla `item` del modelo lógico del TPC-W que se puede ver en la Figura 6.2.

En este capítulo se demostró la posibilidad de desarrollar aplicaciones complejas siguiendo la propuesta *RDBMS-only*, aunque también se advierte que en los primeros pasos casi cada funcionalidad que se pretende desarrollar requiere una extensión a partir del prototipo básico del Capítulo 5. Por este motivo, no es esperable que el producto *webpg* sirva en su estado actual para desarrollar aplicaciones arbitrariamente complejas, y debería estar claro que el objetivo de estos experimentos fue demostrar la viabilidad de la propuesta. En el próximo capítulo se pretende juzgar la propuesta en comparación con otra arquitectura bien establecida, aunque para esta comparación se deben separar las limitaciones inherentes de la propuesta *RDBMS-only* de las limitaciones accidentales debidas al estado actual de implementación de *webpg*. Con todo, se advierte que los juicios son difíciles.

Capítulo 7

Análisis comparativo y evaluación del enfoque propuesto

Y ¿quién eres tú, que pretendes
sentarte en el tribunal para
juzgar a mil millas de distancia,
con una vista que solo alcanza
un palmo?

La divina comedia, el paraíso,
canto XIX
Dante Alighieri

En este capítulo se presenta un análisis comparativo de la tecnología presentada en el Capítulo 6 respecto de otras tecnologías de utilización extendida. Este análisis no pretende emitir un juicio, sino que se presenta como *una forma* sistemática de comparar la arquitectura *RDBMS-only* con las bien establecidas arquitecturas de múltiples niveles. Vale aclarar que *"sistemática"* no implica *"exhaustiva"*, y las comparaciones que aquí se presentan están lejos de ser exhaustivas[1].

Existen múltiples factores de éxito para una aplicación de *e-commerce*, muchos de los cuales están relacionados al negocio y son independientes de la

[1] En otras palabras: la comparación es sistemática porque sigue un sistema, pero no es exhaustiva porque no agota todas las posibles características que se podrían considerar.

tecnología, como la orientación al cliente, la estrategia comercial, la variedad de la oferta, la facilidad de uso, los precios, los sistemas de pago, los envíos, el diseño, el marketing, y la confianza y fidelidad de los clientes [107]. En un análisis como el presente, donde el objeto de estudio es la arquitectura, nada se puede decir sobre estos factores, y el análisis se limita a las consideraciones de arquitectura.

En la búsqueda de un marco estándar para realizar un análisis comparativo de los productos generados con *webpg* y sus alternativas, se eligió la norma ISO 25010 [108]. Esta norma ofrece un modelo para realizar una evaluación sobre calidad del producto de software, y prescribe ocho características de calidad, cada una con subcaracterísticas, como se puede apreciar en la Tabla 7.1. Existen algunas características como Adecuación funcional y Usabilidad, que son independientes de la arquitectura, de las cuales nada diremos.

Característica	**Subcaracterística**	**Análisis**
Adecuación funcional	Completitud funcional	No
	Corrección funcional	No
	Pertinencia funcional	No
Usabilidad	Inteligibilidad	No
	Aprendizaje	No
	Operabilidad	No
	Protección frente a errores de usuario	No
	Estética	No
	Accesibilidad	No
Compatibilidad	Coexistencia	Conceptualmente
	Interoperabilidad	Conceptualmente
Fiabilidad	Madurez	Conceptualmente
	Disponibilidad	Conceptualmente
	Tolerancia a fallos	Conceptualmente
	Capacidad de recuperación	Conceptualmente
Seguridad	Confidencialidad	Conceptualmente
	Integridad	Conceptualmente
	No repudio	Conceptualmente
	Autenticidad	Conceptualmente
	Responsabilidad	Conceptualmente

Característica	**Subcaracterística**	**Análisis**
Mantenibilidad	Modularidad	Conceptualmente
	Reusabilidad	Conceptualmente
	Capacidad de ser analizado	Conceptualmente
	Capacidad de ser modificado	Conceptualmente
	Capacidad de ser probado	Conceptualmente
Portabilidad	Adaptabilidad	Conceptualmente
	Facilidad de instalación	Conceptualmente
	Capacidad de ser reemplazado	Conceptualmente
Desempeño	Comportamiento temporal	En profundidad
	Utilización de recursos	En profundidad
	Capacidad	En profundidad

Tabla 7.1: Modelo de calidad del producto de ISO/IEC 25010

Se discutirán las otras seis características, aunque para mantener acotado el alcance, este análisis comparativo se mantendrá a nivel conceptual para las características Compatibilidad (en la Sección 7.1), Fiabilidad (en la Sección 7.2), Seguridad (en la Sección 7.3), Portabilidad (en la Sección 7.4) y Mantenibilidad (en la Sección 7.5). Esta comparación conceptual se realizará entre el prototipo *webpg* propuesto en el Capítulo 5, y una aplicación teórica en tres niveles que utilice *Java Servlets* para el nivel medio y un RDBMS para el nivel de persistencia. Llamaremos $\mathrm{T}_{\mathrm{webpg}}$ a la tecnología del prototipo *webpg*, y $\mathrm{T}_{\mathrm{java}}$ a la tecnología que utiliza *Java Servlets*.

Solo para la característica Desempeño se realiza un análisis detallado, que incluye pruebas de desempeño, en la Sección 7.6.

Finalmente, la Sección 7.7 presenta un resumen del análisis realizado.

7.1. Compatibilidad

Esta característica se define como la capacidad de dos o más sistemas o componentes para intercambiar información y/o llevar a cabo sus funciones requeridas cuando comparten el mismo entorno de hardware o software. Esta característica se subdivide a su vez en las siguientes subcaracterísticas:

- **Coexistencia**. Capacidad del producto para coexistir con otro software independiente, en un entorno común, compartiendo recursos comunes sin

detrimento.

- **Interoperabilidad**. Capacidad de dos o más sistemas o componentes para intercambiar información y utilizar la información intercambiada.

En términos de **coexistencia**, las dos tecnologías podrían funcionar sobre sistemas operativos UNIX, Linux y Windows, T_{webpg} supone la existencia de un servidor web Apache y un RDBMS PostgreSQL, mientras que T_{java} supone la existencia de un *Servlet Container* y un RDBMS cualquiera. Esto podría suponer un argumento a favor de alguna de las dos tecnologías, en caso que se contara previamente con alguno de los componentes de software de base requeridos, pero no supone ninguna ventaja *a priori* para ninguna de las dos tecnologías en un análisis independiente del contexto. T_{webpg} no necesita que el software de base sea dedicado, y puede coexistir con otras aplicaciones que utilicen el mismo servidor web Apache y el mismo RDBMS PostgreSQL. T_{java} tampoco necesita que el software de base sea dedicado, y puede coexistir con otras aplicaciones que utilicen el mismo *Java Servlet Container* y el mismo RDBMS. Las dos tecnologías tendrían una evaluación favorable y similar con respecto a coexistencia.

En términos de **interoperabilidad** T_{java} tiene una gran ventaja, ya que existen numerosas bibliotecas que permiten interoperar con otros sistemas mediante *Web Services* si se utiliza un *Java Servlet Container*, y mediante *Java Message Service* (JMS), *Java EE Connector Architecture* (JCA) y EJB si se utiliza un *Java EE Application Server*. Vale destacar que estas últimas posibilidades de interoperabilidad de T_{java} son aplicables cuando el otro sistema también utiliza un *Java EE Application Server*.

En T_{webpg} podrían desarrollarse bibliotecas similares, pero hay que considerar que no están implementadas. APEX, por mencionar las capacidades de interoperabilidad de una tecnología que sigue una arquitectura similar, ha implementado el soporte de *REST Web Services* en el componente ORDS así como soporte para *OAuth 2.0* y *OpenID Connect* (OIDC) para poder interoperar con cualquier sistema de *Single Sign-On* que utilice alguno de estos estándares.

Podemos concluir que la compatibilidad de T_{java} es similar a la de T_{webpg} en términos de coexistencia, pero que la compatibilidad de T_{java} es superior a la de T_{webpg} en términos de interoperabilidad, considerando su estado actual de desarrollo como prototipo, y que se necesitaría mucho desarrollo para que

T_{webpg} pudiera llegar a tener las posibilidades de interoperabilidad de T_{java}.

7.2. Fiabilidad

Esta característica se define como la capacidad de un sistema o componente para desempeñar las funciones especificadas, cuando se usa bajo unas condiciones y período de tiempo determinados. Esta característica se subdivide a su vez en las siguientes subcaracterísticas:

- **Madurez**. Capacidad del sistema para satisfacer las necesidades de fiabilidad en condiciones normales.
- **Disponibilidad**. Capacidad del sistema o componente de estar operativo y accesible para su uso cuando se requiere.
- **Tolerancia a fallos**. Capacidad del sistema o componente para operar según lo previsto en presencia de fallos de hardware o software.
- **Capacidad de recuperación**. Capacidad del producto de software para recuperar los datos directamente afectados y reestablecer el estado deseado del sistema en caso de interrupción o fallo.

En términos de **madurez**, tanto T_{webpg} como T_{java} tienen un cierto nivel de madurez derivado de la madurez del software de base subyacente. En condiciones normales, podría argumentarse que un servidor web Apache y un RDBMS PostgreSQL tienen una madurez al menos tan alta como cualquier *Java Servlet Container* o *Java EE Application Server* y cualquier otro RDBMS. Sin embargo, T_{webpg} se debería someter a pruebas de uso en entornos de producción, y a una evolución prolongada en el tiempo, ya que no pueden considerarse maduros ni el *web listener* ni la *engine*. Mientras esto no ocurra, es razonable concluir que T_{java} tiene un mayor nivel de madurez, no debido a alguna característica intrínseca de la arquitectura sino a la escasa madurez del prototipo.

En términos de **disponibilidad** y **tolerancia a fallos**, en el web server tanto un servidor Apache como la mayoría de los *Java Servlet Containers* o *Java EE Application Servers* pueden configurarse en modalidad de *cluster* o dominio para ofrecer alta disponibilidad ante un fallo de hadrware o software en uno de los *hosts*. Incluso podrían configurarse como PODS sobre una tecnología de contenedores como *kubernetes* [109]. En el nivel de persistencia,

PostgreSQL (como la mayoría de los RDBMSs maduros) puede configurarse en un entorno de alta disponibilidad ante un fallo de hadrware o software en uno de los *hosts*, por ejemplo utilizando una base *standby* con *Streaming Replication* y alguna forma de *failover* automático utilizando productos como *repmgr* [110] y/o *pgpool* [99]. Esta configuración de *cluster* activo-pasivo podría suponer, ante una caída del *host* con la base primaria, que pueden fallar los *requests* que lleguen entre la caída y la compleción del *failover* (esto es, la promoción de la base *standby* como primaria y la redirección de los *requests* desde el *web server*), período que típicamente puede llevar unos pocos segundos pero puede variar dependiendo del diseño de la solución y la configuración de los componentes involucrados. Si bien podría argumentarse que otros RDBMSs tienen soluciones nativas de *cluster* activo-activo, también es cierto que en algunos casos implican licenciamiento especial con costo extra (como en el caso de *Oracle Real Application Clusters*) y que T_{webpg} es portable a cualquier RDBMS con capacidades de *cluster* activo-activo basado en PostgreSQL como *EnterpriseDB/Postgres-BDR* [111]. Vale destacar además que los *clusters* de RDBMSs de tipo activo-activo suelen agregar una gran complejidad de configuración y administración a una solución. En resumen, las dos arquitecturas podrían lograr muy buenos niveles de disponibilidad y tolerancia a fallos si se configura adecuadamente el software de base.

En términos de **capacidad de recuperación**, es necesario distinguir los dos tipos de fallos de los que es posible tener que recuperarse: fallos de disponilidad de hardware o software, y fallos que generen problemas lógicos de datos. Para recuperarse de un fallo de disponibilidad de hardware o software, podría aplicar lo discutido en el párrafo anterior, si se tomaron medidas para ofrecer alta disponibilidad. Si estas medidas no se tomaron, los mecanismos de T_{webpg} y T_{java} son similares, ya que implican la restauración de un respaldo de los *hosts* del nivel medio, y la restauración a un punto en el tiempo de la base de datos, a partir de respaldos y *logs* transaccionales archivados, utilizando las funcionalidades del RDBMS, que son similares entre PostgreSQL y cualquier otro RDBMS. Vale aclarar que en arquitecturas complejas de N niveles, los "*hosts* del nivel medio" podrían ser múltiples, y en este caso podría tenerse una ventaja en T_{webpg}. Para recuperarse de una falla lógica (por ejemplo, se elimina una tabla a causa de un *bug* de software, un ataque o un error de un administrador) se debe identificar el momento de la falla y proceder a una restauración a un punto en el tiempo de la base de datos, a partir de respaldos

y *logs* transaccionales archivados.

Podemos concluir que la fiabilidad de T_{webpg} es comparable a la de T_{java} en las subcaracterísticas de disponibilidad, tolerancia a fallos y capacidad de recuperación; pero que la fiabilidad de T_{java} es superior a la de T_{webpg} en la subcaracterística de madurez, debido al estado de desarrollo del prototipo.

7.3. Seguridad

Esta característica se define como la capacidad de protección de la información y los datos de manera que personas o sistemas no autorizados no puedan leerlos o modificarlos. Esta característica se subdivide a su vez en las siguientes subcaracterísticas:

- **Confidencialidad**. Capacidad de protección contra el acceso de datos e información no autorizados, ya sea accidental o deliberadamente.
- **Integridad**. Capacidad del sistema o componente para prevenir accesos o modificaciones no permitidos a datos o programas.
- **No repudio**. Capacidad de demostrar las acciones o eventos que han tenido lugar, de manera que dichas acciones o eventos no puedan ser repudiados posteriormente.
- **Responsabilidad**. Capacidad de rastrear de forma inequívoca las acciones de una entidad.
- **Autenticidad**. Capacidad de demostrar la identidad de un sujeto o un recurso.

En términos de **confidencialidad**, no hay diferencias significativas en el nivel medio entre T_{webpg} y T_{java}, salvo las que puedan deberse a la falta de madurez de T_{webpg}. Por ejemplo, no se tomaron medidas para prevenir un *session hijacking*, aunque no hay ninguna razón inherente a la arquitectura o a la tecnología por la cual no puedan implementarse. Al nivel de la persistencia, PostgreSQL cuenta con la capacidad de utilizar conexiones sobre *Secure Sockets Layer* (SSL), tiene su propio sistema de control de accesos en el archivo de *host based authentication* llamado `pg_hba.conf`, cuenta con un esquema de permisos similar al del resto de los RDBMSs e incluye la posibilidad de implementar cifrado a nivel de columnas específicas de una tabla o a nivel de archivos para aumentar la seguridad de la autenticación y

autorización. Algunas de estas capacidades (como la de cifrar una columna en una tabla) no son transparentes para las aplicaciones. Otros RDBMS como Oracle tienen capacidades mayores, y mediante el uso de *Advanced Security Option* (ASO) permiten utilizar funcionalidades de *Transparent Data Encryption* y *Data Redaction* que ofrecen una mejor usabilidad de similares características de seguridad. Vale mencionar que en cuanto a *SQL Injection*, una de las principales vulnerabilidades de las aplicaciones web, T_{webpg} puede ser superior a T_{java} según cómo se hayan implementado las sentencias en T_{java}. El *SQL Injection* es una posibilidad en la medida que se utiliza SQL dinámico, lo que está prohibido en T_{webpg} por diseño, ya que los accesos a los datos nunca se realizan mediante sentencias SQL sino mediante los *Stored Procedures* del *DL Code*, según se presentó en el Capítulo 4. En T_{java} los accesos podrían ser vulnerables a un *SQL Injection* si se realizan mediante SQL dinámico (objetos *Statement*), o podrían no serlo si se realizan mediante SQL estático (objetos *PreparedStatement* y *CallableStatement*) en su totalidad.

En términos de **integridad**, se puede decir que no hay grandes diferencias entre T_{webpg} y T_{java}, ya que ambas tecnologías utilizan un RDBMS con capacidad de definir restricciones de integridad mediante SQL. Más allá de la integridad referencial, la eventual superioridad de T_{webpg} sobre T_{java} por prevenir el *SQL Injection* por diseño, podría constituir un argumento de mayor seguridad en las subcaracterísticas de confidencialidad e integridad.

En términos de **no repudio**, cabe distinguir entre dos casos. El primero, la posibilidad de generar pistas de auditoría para las modificaciones de datos sensibles, donde se registre el usuario, el momento y las versiones de las modificaciones (datos previos y posteriores). En PostgreSQL es posible implementar esto en parte mediante configuración y en parte mediante triggers. Otros RDBMSs como Oracle tienen características como *Audit Trail* y *Fine Grained Auditing* que mejoran las posibilidades de auditar eventos específicos. Estas características podrían implementarse en T_{webpg} en el *DL Code* como fue discutido en la Sección 6.3. En este caso, tanto en T_{webpg} como en T_{java}, se tiene el problema del mantenimiento del usuario nominado, ya que el usuario que está efectivamente conectado al RDBMS es un usuario genérico que mantiene un *pool* de conexiones. También existen formas de solucionar este problema tanto en T_{webpg} como en T_{java}, aunque la posibilidad de realizarlo en T_{webpg} se postula de forma teórica y no se realizó ninguna prueba de concepto para validarla. El segundo caso de no repudio, lo podría constituir la posibilidad de

agregar una firma electrónica avanzada a alguna modificación, lo que constituye una funcionalidad a implementar en el nivel del cliente (por ejemplo mediante *WebSockets* que se comuniquen con un software de gestión de firmas del lado del cliente, o con un *token* o lector de tarjetas inteligentes). En este caso, las capacidades de T_{webpg} y T_{java} son equivalentes, salvo por las consideraciones de madurez que pueden implicar la existencia de bibliotecas específicas en el caso de T_{java} que no estarían disponibles para T_{webpg}.

En términos de **responsabilidad**, las capacidades de T_{webpg} y T_{java} son equivalentes. En ambos casos se tienen similares capacidades de auditoría, y se tiene el problema del mantenimiento del usuario nominado discutido en el párrafo anterior, inherente al uso de un *pool* de conexiones.

7.4. Portabilidad

Esta característica se define como la capacidad del producto o componente de ser transferido de forma efectiva y eficiente de un entorno hardware, software, operacional o de utilización a otro. Esta característica se subdivide a su vez en las siguientes subcaracterísticas:

- **Adaptabilidad**. Capacidad del producto que le permite ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de hardware, software, operacionales o de uso.
- **Capacidad para ser instalado**. Facilidad con la que el producto se puede instalar y/o desinstalar de forma exitosa en un determinado entorno.
- **Capacidad para ser reemplazado**. Capacidad del producto para ser utilizado en lugar de otro producto de software determinado con el mismo propósito y en el mismo entorno.

En términos de **adaptabilidad** al entorno de hardware y sistema operativo cabría decir que no hay diferencias significativas entre T_{webpg} y T_{java}: en ambos casos un producto desarrollado en estas tecnologías podría adaptarse a múltiples entornos de hardware y sistema operativo. Las dos tecnologías podrían considerarse igualmente aptas para entornos *on premise* y *on demand.* Donde sí existe una diferencia en la capacidad de adaptación es en las tecnologías de software de base: mientras que podría haber alguna pequeña

dificultad en adaptar T_{java} a un entorno diferente de RDBMS y servidor de aplicaciones, T_{webpg} es altamente dependiente del RDBMS por cuanto no existe un DBPL estándar.

En términos de **capacidad para ser instalado** tanto T_{webpg} como T_{java} son tecnologías muy fáciles de instalar. T_{java} implica la instalación de un RDBMS, la creación de una base, la instalación de un Tomcat, un driver JDBC, el despliegue de una aplicación y unos pocos pasos de configuración. T_{webpg} implica la instalación de un RDBMS, la creación de una base, la instalación de un *web server* Apache con un módulo, una biblioteca de libpq y unos pocos pasos de configuración.

La **capacidad para ser reemplazado** es un poco más compleja de analizar, porque en general esta capacidad depende de cuál sea el otro producto en cuestión. Considerando la capacidad de reemplazar un producto como el del Capítulo 6 entre T_{webpg} y T_{java}, podemos decir que tienen similares capacidades de ser reemplazados el uno por el otro, pero este reemplazo implica una reimplementación. La dificultad de la reimplementación en T_{webpg} dependerá de la existencia y facilidades de un IDE.

7.5. Mantenibilidad

Esta característica se define como la capacidad del producto de software para ser modificado efectiva y eficientemente, debido a necesidades evolutivas, correctivas o perfectivas. Esta característica se subdivide a su vez en las siguientes subcaracterísticas

- **Modularidad**. Capacidad de un sistema o programa de ordenador (compuesto de componentes discretos) que permite que un cambio en un componente tenga un impacto mínimo en los demás.
- **Reusabilidad**. Capacidad de un activo que permite que sea utilizado en más de un sistema de software o en la construcción de otros activos.
- **Capacidad de ser analizado**. Facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el software, o identificar las partes a modificar.
- **Capacidad para ser modificado**. Capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o

degradar el desempeño.

- **Capacidad para ser probado**. Facilidad con la que se pueden establecer criterios de prueba para un sistema o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.

En términos de **modularidad** puede argumentarse que no hay grandes diferencias entre T_{webpg} y T_{java}, ya que en ambas tecnologías es posible el diseño de software en base a unidades modulares. Más allá de las diferencias de paradigma de programación, casi toda organización modular que puede diseñarse en T_{java} a través de las construcciones de *class* y *package*, puede diseñarse en T_{webpg} a través de las construcciones de *function* y *schema*. La diferencia fundamental es que la programación estructurada no tiene el concepto de herencia, y por tanto no existe un equivalente a los modificadores *protected*. Otra diferencia es que el control del acceso a cada unidad modular, en lugar de declararse en la propia unidad modular como en T_{java}, se logra a través del control de los privilegios con sentencias `grant`, `revoke` y `alter default privileges`. Vale mencionar que PostgreSQL es un *Object-Relational Database Management System* (ORDBMS) y como tal ofrece la posibilidad de utilizar herencia, pero no utilizaremos esta caacterística como argumento ya que la elección de PostgreSQL no estuvo basada ella, y esta discusión pretende tener una validez más general aplicable a todo RDBMS que tenga un DBPL razonablemente desarrollado. Por esto, podría decirse que, en general, T_{java} y T_{webpg} son similares en cuanto a su capacidad de producir software mantenible a través de un diseño modular, salvo por la ventaja de T_{java} de ofrecer las posibilidades de la herencia y el polimorfismo.

En términos de **reusabilidad**, ambas tecnologías permiten de forma similar la reutilización *intratecnológica* (es decir, la reutilización de código *dentro* de la misma tecnología). Una diferencia fundamental es que el código de T_{java}, al ser Java un estándar, podría desplegarse con relativa facilidad en cualquier otro *Servlet Container*, mientras que el código T_{webpg}, al no existir un DBPL estándar, supone un costo de traducción para un eventual reuso en otro RDBMS. Asumiendo que en determinados casos es posible que la tecnología de la interfaz deba cambiar más rápido que la de la lógica del negocio, el paradigma de ThickDB permitiría el reuso del *DL Code* y *BL Code* aún cuando se quisiera pasar de una arquitectura *RDBMS-only* a una

arquitectura *database-centric* con la capa de la interfaz fuera del RDBMS. Sobre esta hipótesis, podríamos decir que la tecnología T_{webpg} tiene alguna ventaja respecto de T_{java} en cuando a la capacidad de reuso.

En términos de **capacidad de ser analizado**, la inexistencia de un IDE en T_{webpg} supone una desventaja muy importante, pero asumiendo la existencia de un IDE, no deberían existir diferencias importantes entre un software desarrollado con T_{webpg} y otro desarrollado con T_{java} en cuanto a su capacidad de ser analizado.

En términos de **capacidad para ser modificado**, vale la misma apreciación que en el párrafo anterior: sin un IDE, la capacidad de modificar un software desarrollado con T_{webpg} es muy baja; pero sin embargo, con un IDE, no deberían existir diferencias importantes entre un software desarrollado con T_{webpg} y otro desarrollado con T_{java} en cuanto a su capacidad de ser modificado. Incluso se puede mencionar la ventaja de que la modificación solo requiere de un *web browser*, lo que podría facilitar la capacidad de modificar el software al evitar cualquier tipo de problemas de ambiente de desarrollo del lado del cliente.

En términos de **capacidad para ser probado**, valdría la pena distinguir entre los casos de prueba que se ejecutan directamente desde la interfaz y los que pueden ejecutarse a partir de la invocación de las construcciones del código. En los casos de prueba que se ejecuten directamente desde la interfaz no hay diferencias entre un software desarrollado con T_{webpg} y otro desarrollado con T_{java}. En los casos de prueba que se ejecuten a partir de la invocación de las construcciones del código, lo cual podría permitir la automatización de casos de prueba de forma muy simple, en el caso de T_{webpg} se tiene la ventaja de poder implementarlos y ejecutarlos únicamente a través de un *web browser*, lo que podría suponer una ventaja.

7.6. Desempeño

Esta característica representa el desempeño relativo a la cantidad de recursos utilizados bajo determinadas condiciones. Esta característica se subdivide a su vez en las siguientes subcaracterísticas:

- **Comportamiento temporal**. Los tiempos de respuesta y procesamiento y las tasas de rendimiento (*throughput*) de un

sistema cuando lleva a cabo sus funciones bajo condiciones determinadas en relación con un banco de pruebas (*benchmark*) establecido.

- **Utilización de recursos**. Las cantidades y tipos de recursos utilizados cuando el software lleva a cabo su función bajo condiciones determinadas.
- **Capacidad**. Grado en que los límites máximos de un parámetro de un producto o sistema software cumplen con los requisitos.

Los SUTs para las comparaciones de desempeño serán, por un lado, la implementación de Wisconsin en Java Servlets del TPC-W, y por otro, la implementación en *webpg* del Capítulo 6.

Para esta característica de *desempeño* se realizaron pruebas y se presentan los resultados de las mismas. Por este motivo, es preciso contar con definiciones más precisas que las ofrecidas por la norma ISO 25010. Si bien puede haber múltiples definiciones de los términos desempeño y escalabilidad, no se realizará un análisis de las cuestiones disputadas sobre estos temas y utilizaremos las siguientes definiciones [112, 113].

Definición 7.1 *Trabajo es el conjunto de operaciones requeridas para completar una tarea computacional*

Por ejemplo, en el contexto de un sistema de *e-commerce*, un trabajo podría ser "Agregar un ítem a un carrito de compras" (le llamaremos trabajo A) o "Cambiar el medio de pago de un cliente" (le llamaremos trabajo C). En aplicaciones web, de gestión y transaccionales, es común asociar el trabajo a una transacción en la base de datos.

Definición 7.2 *Desempeño (como tiempo de respuesta) es el tiempo que un cliente observa desde que realiza la solicitud de un trabajo hasta que tiene la respuesta*

El tiempo de respuesta de dos tipos de trabajo pueden ser diferentes por naturaleza, ya que pueden requerir diferentes tipos de accesos a los recursos computacionales. El tiempo de respuesta de dos ejecuciones del mismo trabajo podría ser diferente debido a una variabilidad normal de la disponibilidad de los recursos en sistemas de tiempo compartido, pero debería tender a un valor estable, característico del trabajo, como promedio de un número grande de repeticiones. Un trabajo A podría tener un tiempo de respuesta característico de T_A y un trabajo C un tiempo de respuesta característico de T_C.

Definición 7.3 *Desempeño (como carga) es una medida de cantidad de trabajo por unidad de tiempo*

Si se consideran los trabajos A y C es posible que una determinada carga de trabajo realice N_A trabajos A y N_C trabajos C por unidad de tiempo. Si los trabajos se identifican con transacciones, el desempeño como carga podría medirse en términos de transacciones por unidad de tiempo, típicamente transacciones por segundo (tps). Un sistema podría tener un desempeño simultáneo de N_A transacciones A por segundo y N_C transacciones C por segundo. Si la relación entre N_A y N_C es característica del sistema, es posible simplificar la métrica diciendo que el sistema tiene un desempeño de (N_A + N_C) tps.

Definición 7.4 *Escalabilidad es la capacidad de un sistema de adaptarse a un incremento de la carga de trabajo, bien sin afectar el desempeño como tiempo de respuesta, o bien aumentando en la misma proporción el desempeño como carga. Eventualmente, esta capacidad puede ser conseguida agregando recursos computacionales al sistema.*

Por ejemplo, podríamos duplicar la cantidad de solicitudes de trabajos A y C por unidad de tiempo, y verificar si el desempeño como tiempo de respuesta sigue siendo T_A y T_C respectivamente, o bien verificar si el desempeño del sistema también se duplicó a 2 * (N_A+ N_C) tps. Típicamente, los sistemas pueden escalar hasta cierto punto a partir del cual se comienza a observar una degradación, y es posible graficar el desempeño (en cualquiera de sus acepciones) en función de la cantidad de solicitudes por unidad de tiempo. Esta función, característica de un sistema, puede utilizarse para comparar un sistema con otros en términos de desempeño y escalabilidad.

Las definiciones precedentes se utilizarán en los resultados de las pruebas de desempeño. Se realizaron pruebas siguiendo un *benchmark* representativo para sistemas de *e-commerce* del TPC, organización que publica *benchmarks* que modelan la carga típica de diferentes tipos de sistemas. Para estas pruebas de desempeño, utilizamos un *benchmark* inspirado en *TPC Benchmark™ W* (TPC-W) [101]. El *benchmark* TPC-W modela el sitio web de una librería como sistema típico de *e-commerce*, que es una aplicación web (Definición 2.1), de gestión (Definición 2.2) y transaccional (Definición 2.3), según fueron definidas en la Sección 2.1. Este *benchmark*, ampliamente reconocido en la literatura

[103, 114, 115, 102], reproduce la carga generada por múltiples sesiones de *web browsers* sobre una aplicación web, que sirve contenido estático y dinámico. TPC-W ofrece un ambiente estándar independiente de la tecnología subyacente, la arquitectura diseñada y la infraestructura desplegada. Para poblar la base se utilizó una implementación de código abierto en Java del TPC-W con ligeras modificaciones para adaptarlo de DB2 a PostgreSQL [104]. El modelo lógico generado se puede apreciar en la Figura 6.2.

La Figura 7.1 muestra lo que las dos implementaciones comparten y cómo difieren. Para esta comparación utilizamos la misma base de datos, con el mismo esquema y el mismo volumen de datos. La implementación T_{java} accede a la base de datos a través de JDBC y obtiene las imágenes del *filesystem*. La implementación T_{webpg} utiliza el módulo `mod_plpgsql` como un adaptador entre los protocolos HTTP y *libpq*, pero realiza todo el procesamiento de los *requests* en el RDBMS, incluyendo la recuperación de las imágenes que se mantienen en tablas.

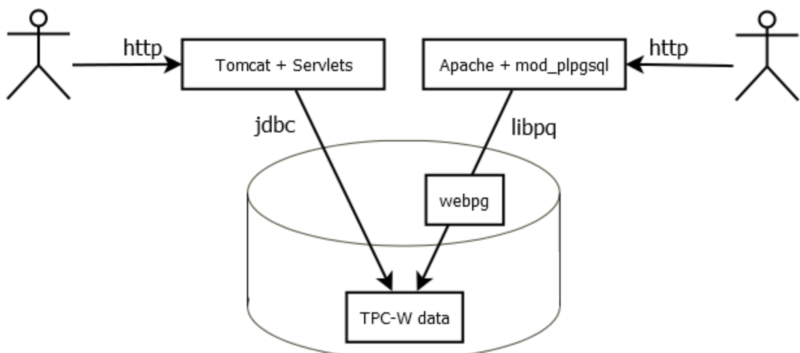


Figura 7.1: Accesos mediante Wisconsin y webpg

Es importante notar que se están comparando dos sistemas que tienen la misma interfaz, el mismo comportamiento, el mismo RDBMS y los mismos datos. La diferencia radica en la arquitectura, mientras T_{java} sigue la bien establecida arquitectura de tres niveles de tipo *thin/fat/thin*, T_{webpg} sigue una arquitectura *RDBMS-only* de tipo *thin/zero/fat*.

Los tests de desempeño del TPC-W prescriben tres tipos de escenarios de uso en un sitio de *e-commerce*: *Shopping*, *Browsing* y *Ordering*. Para cada uno de estos escenarios, se establece la frecuencia de *requests* para cada una de las 14 páginas del sitio. La medida de desempeño más relevante es el *Web Interaction Response Time* (WIRT), que es el tiempo que pasa desde que el

primer *byte* es enviado desde el *browser* hasta que el *browser* recibe el último byte.

La Figura 7.2 presenta los resultados de una comparación de peformance utilizando la métrica WIRT. Las frecuencias de tiempos de carga para dos de las páginas más accedidas fueron comparadas durante una interacción de tipo *Shopping Mix*.

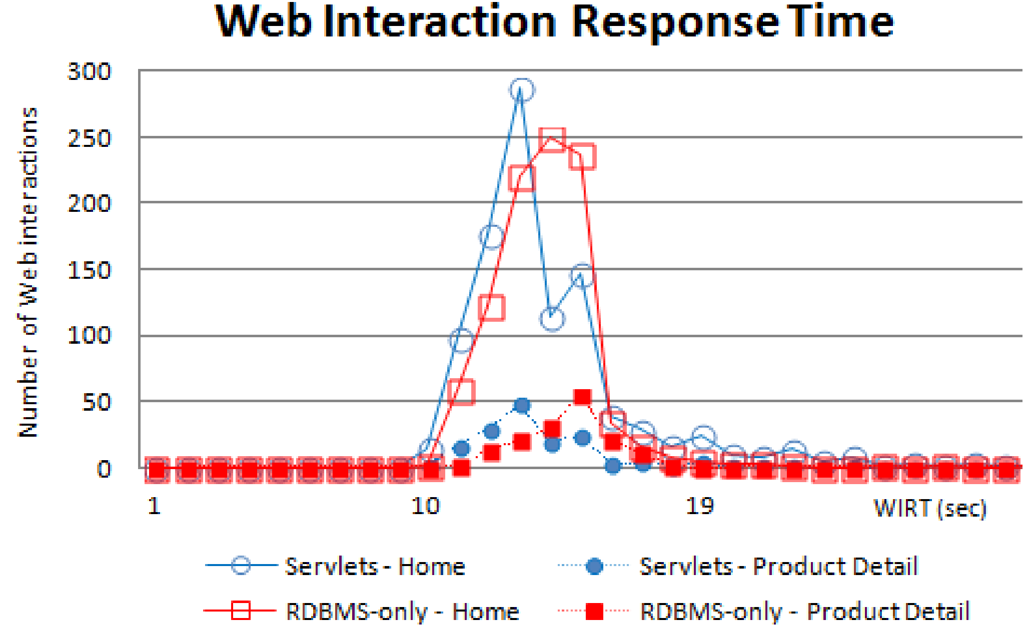


Figura 7.2: Comparación de desempeño

La Figura 7.2 nuestra que no hay diferencias significativas de desempeño entre las dos implementaciones. Tampoco hay diferencias significativas entre esta comparación y otras que incluyan las demás páginas y otros tipos de interacción (*Browsing* y *Ordering*). Una observación preliminar es que la implementación T_{java} tiene una frecuencia más alta de tiempos de respuesta ligeramente mejores, pero tiene una dispersión más alta que la de la implementación T_{webpg}. Esta última parece tener tiempos de respuesta ligeramente mayores, pero presenta menos dispersión y es más estable en los tiempos de respuesta observados. Vale destacar que en ninguno de los casos se realizó ningún trabajo de optimización en la base de datos.

Otro aspecto interesante en cuanto a desempeño es la diferencia en el proceso de carga de una página a nivel del protocolo HTTP. Las figuras 7.3 y 7.4 presentan capturas de pantalla de la pestaña *Network request list* de la

funcionalidad *Network Monitor* de Firefox [116], y permiten ver un detalle de los tiempos de carga de los diferentes recursos solicitados por HTTP. Como puede verse en la Figura 7.3, la carga de una página en T_{java} comienza por un *request* al *Servlet*, que se resuelve relativamente rápido, pero el código de la página que retorna hace referencia a otros recursos, como las imágenes, que generan *subrequests*.

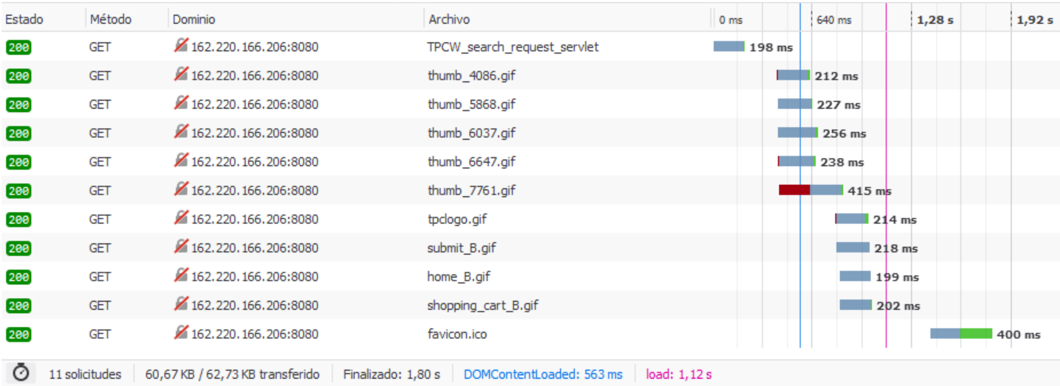


Figura 7.3: Carga de una página en T_{java}

En el caso de T_{webpg}, como puede verse en la Figura 7.4, el *request* a una página es único y no genera otros *subrequests*.

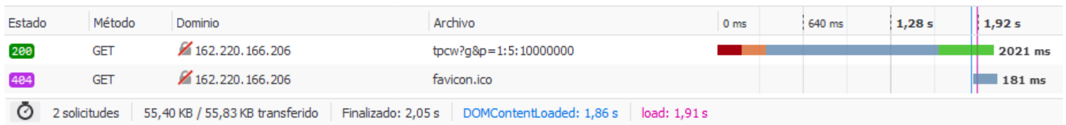


Figura 7.4: Carga de una página en T_{webpg}

En resumen, los tiempos de repuesta en T_{webpg} parecen ser un poco mayores, y la mayor parte del tiempo de carga se debe a la espera del *web browser* porque el *web server* le comience a enviar el *HTTP Response*. En el RDBMS, esta demora es fundamentalmente el tiempo de ejecución de la función `g`, que incluye la construcción de toda la página y las sustituciones de las imágenes, que deben recuperarse de la tabla de imágenes y sustituirse en el resultado final.

7.7. Resumen del análisis

En términos generales, las dos arquitecturas consideradas de forma teórica son comparables en todas las características y subcaracterísticas analizadas.

Se advierten serias desventajas cuando se consideran las tecnologías concretas, lo cual se explica por la inmadurez propia de T_{webpg}. Los dos niveles de análisis aportan conclusiones diferentes e interesantes. Por un lado, no sería prudente considerar la implementación de un proyecto real en una tecnología como T_{webpg}, fundamentalmente por su baja fiabilidad. Por otro lado, se debe reconocer que estas limitaciones se deben principalmente a la falta de investigación y desarrollo en esta línea de trabajo. El caso de APEX (no analizado arriba) estaría en un punto intermedio: en términos de fiabilidad (madurez, disponibilidad, tolerancia a fallos y capacidad de recuperación) podría ser al menos tan bien calificado como T_{java}, pero en aspectos como compatibilidad (y específicamente interoperabilidad), T_{java} ofrece una gama de posibilidades que exceden las de APEX. Con todo, dado un proyecto específico habría que analizar si esta diferencia es relevante a la luz de los requerimientos concretos de interoperabilidad que se hayan relevado.

Capítulo 8

Conclusiones y trabajo a futuro

> Muchas cosas sabe el zorro, pero
> el erizo sabe una sola y grande.
>
> *Arquíloco*

En este capítulo se presentan las conclusiones de este trabajo y se delinean algunos de los problemas a abordar para continuar avanzando en esta línea de investigación.

La motivación inicial de este trabajo surgió a partir del conocimiento de APEX. Esta tecnología podría representar una anomalía en el sentido de seguir una arquitectura que no satisface las expectativas sobre la forma en que *debería* estructurarse una aplicación web. Esas expectativas generalmente son tácitas, y surgen de la exposición continua a trabajos, libros de texto y tecnologías que proponen, describen y siguen arquitecturas muy diferentes.

La relación que existe entre APEX y la arquitectura *RDBMS-only* es del tipo ejemplar-especie, con la particularidad de ser además el único ejemplar conocido de la especie. A efectos académicos, se puede justificar que la arquitectura *RDBMS-only* tiene interés, al ser una especie de arquitectura que no ha sido descrita, ni aún para refutarla como arquitectura válida. Por otra parte, la experiencia muestra con más fuerza que cualquier análisis teórico, que la arquitectura es útil en el sentido de adaptarse bien al uso, al menos en algunos escenarios como los presentados y en otros conocidos de la industria incluyendo algunos de los sitios web de la propia *Oracle Corporation* como el *Oracle Store* [117]. Considerando todo esto, parecía sensato comenzar analizando la arquitectura interna de APEX tanto como

fuera posible, no con interés tecnológico, sino tratando de obtener lo general de la especie a partir de lo concreto del ejemplar, tratando de discernir lo necesario de lo contingente. Ese análisis es el que se presentó en el Capítulo 3. Era de esperar que de este análisis pudiera obtenerse la descripción de una arquitectura con características estructurales propias y problemas inherentes, con ventajas y escenarios apropiados para su uso así como limitaciones. Esa descripción de la arquitectura es la presentada en el Capítulo 4, pero vale decir que el orden de los capítulos no representa el orden cronológico del trabajo. Para todo el proceso de conceptualización y propuesta, ayudaron mucho la construcción de un nuevo ejemplar de la especie del Capítulo 5 y los experimentos de su utilización del Capítulo 6, lo que permitió enfrentarse a los problemas tecnológicos, contribuyendo así a fijar ideas y analizar la factibilidad de diferentes alternativas de solución para esos problemas. Ahora podría argumentarse que APEX dejó de ser el único ejemplar conocido de una especie no analizada, y que APEX y *webpg* (pese a toda su inmadurez) son dos ejemplares de la arquitectura *RDBMS-only*.

Una pregunta que surge muy fácil después de justificar la pertinencia y oportunidad de un trabajo que no tiene muchos antecedentes, es por qué no los tiene. En otras palabras, por qué aparecen tan pocas referencias a APEX y a una arquitectura como la descrita en la literatura especializada. Si bien la pregunta surge muy fácil, la respuesta parece difícil. Los factores podrían ser múltiples, y algunos podrían encontrarse en dominios tan diferentes, y lejanos para el autor, como la historia y la filosofía de la computación, o aún la psicología. En tal escenario, esbozar una respuesta a esta pregunta, aunque sea parcial, trasciende el alcance de este trabajo y las posibilidades del autor. Sin embargo, se realizaron algunos esfuerzos en ese sentido y creo que puede tener algún valor ofrecer un par de conjeturas que he llegado a formular en el transcurso de este trabajo.

La primera de ellas podría surgir considerando la distinción de Popper de las dos formas de novedad, a saber: la novedad real, de algo intrínsecamente nuevo; y la novedad de arreglos o combinaciones, donde elementos preexistentes se combinan de una nueva manera [118]. Así, algunas postulaciones en computación, como el modelo relacional de Edgar Codd [6, 7, 8, 9, 10] y la Web de Tim Berners-Lee [119] merecen ser llamadas una verdadera novedad; mientras que la idea de Oracle con la arquitectura de APEX es simplemente una novedad de arreglos, en cuanto combina dos tecnologías preeexistentes

(un RDBMS y un *web listener*) y las organiza para funcionar juntas de una forma nueva. Si bien esta novedad de arreglos constituye en la práctica una forma de novedad, no está basada en ninguna idea original publicada. Por otro lado, APEX no fue diseñado desde el comienzo para los objetivos que satisface actualmente, sino que su concepción misma fue incremental. Por supuesto que desde el comienzo se hizo que el RDBMS fuera capaz de atender un *HTTP Request*, con lo cual la arquitectura ya había quedado formulada al menos en la práctica; pero las expectativas de lo que se podría lograr con esta tecnología eran bajas y se limitaban a esperar que pudiera servir para desarrollar reportes y reemplazar planillas de cálculo, con lo que era de esperar que no despertara interés académico. No fue sino con el tiempo y la experimentación que las capacidades de APEX fueron aumentando para permitir mantener aplicaciones con estado y para ofrecer un IDE construido con la misma tecnología. Adicionalmente, la idea de compactar toda una aplicación empresarial en un nivel físico parece ir en el sentido opuesto de la tendencia actual de separación en múltiples niveles, arquitecturas orientadas a servicios y componentes desacoplados para funcionar sobre contenedores. Y finalmente, Oracle tampoco ha tenido la iniciativa de publicar ningún detalle sobre la arquitectura de APEX a nivel académico, y se ha limitado a describir solo los aspectos de la arquitectura que son relevantes para los administradores y desarrolladores de la aplicación, y en sus círculos técnicos (e.g. *Oracle Technology Network*, *Oracle Open World*, foros). De esta forma, tanto por ser meramente una novedad de arreglos como por haberse vuelto lenta y progresivamente una alternativa seria, por ir a contramano de las tendencias, y por el poco interés del proveedor en realizar y publicar trabajos de investigación sobre el tema, es posible que esta arquitectura haya pasado desapercibida para la academia.

La segunda conjetura deriva del hecho de que la web fue inventada con posterioridad a la orientación a objetos. Después de las primeras experiencias con tecnologías como CGI, los objetos fueron una tendencia dominante en las aplicaciones web, y junto con los objetos las arquitecturas de 3 niveles, el *middleware* y las arquitecturas de N niveles. Estas tendencias pudieron ser vistas como un verdadero "paradigma" en el sentido del "paradigma científico" de Kuhn, donde un paradigma nuevo reemplaza al paradigma anterior [120]. Las limitaciones de CGI pudieron ser vistas como evidencia de una crisis del "paradigma de la programación estructurada", y en ese sentido

el nuevo "paradigma de la orientación a objetos" no constituía simplemente una alternativa sino una superación. Estas ideas, explícita o implícitamente, pudieron haber dominado en las comunidades tanto de la academia como de la industria, y quizá podrían explicar la escasez de trabajos enfocados en arquitecturas *database-centric*, y la singularidad de APEX como caso extremo de arquitectura *RDBMS-only*.

Estas conjeturas podrían tomarse como hipótesis para una investigación en los campos de la historia y filosofía de la computación.

Más allá de conjeturas, creo haber demostrado que la arquitectura *RDBMS-only* es una arquitectura válida para sistemas web, de gestión y transaccionales; así como la factibilidad de implementar sistemas de este tipo con cualquier RDBMS que incluya un DBPL razonablemente desarrollado. Asimismo, queda claro que esta arquitectura ofrece algunas ventajas respecto de las clásicas arquitecturas de N niveles, especialmente por la reducción de la complejidad tecnológica [20]. Esta simplificación tecnológica puede redundar en una reducción de los puntos de falla así como en una simplificación de operaciones. En términos de desarrollo, las ventajas o desventajas dependerán mucho de las capacidades del IDE, pero el caso de APEX demuestra que es posible lograr una verdadera plataforma *Low-Code* que permita una alta productividad.

El enfoque ThickDB tal como se presenta en la Figura 2.14 también sugiere una ventaja, derivada de la posibilidad de tener múltiples tecnologías de interfaz sobre un RDBMS que resuelva la lógica de negocio de manera uniforme. Esto podría permitir la evolución de las tecnologías de interfaz sin modificar la lógica de negocio, reduciendo el costo, el tiempo y los riesgos de esa evolución. También se podría argumentar que es esperable que las tecnologías de la capa de presentación cambien con mayor velocidad que las tecnologías de las capas lógica y de acceso a datos.

También podrían existir ventajas dependientes del contexto. Como en el caso de Koppelaars [69], alguien podría tener que tomar una decisión de arquitectura con restricciones, por ejemplo contando con un grupo de desarrollo con un alto nivel de experticia en un DBPL pero no en lenguajes orientados a objetos, y/o con la necesidad de hacer evolucionar un sistema con buena parte o toda la lógica de negocio desarrollada en un DBPL. Sea para aprovechar de la mejor manera los recursos disponibles, o para reutilizar el código de la lógica de negocio, un enfoque *RDBMS-only* podría resultar especialmente ventajoso.

Esta propuesta también presenta algunas desventajas. Algunas son

evidentes, y se derivan de la inexistencia de algunas funcionalidades muy potentes de la orientación a objetos como la herencia y el polimorfismo, así como de las posibilidades de articulación de un trabajo distribuido como puede realizarse en arquitecturas de N capas.

Otros aspectos clave donde esta propuesta necesita ser mejorada son el control de versiones y la gestión de cambios. Las metodologías ágiles requieren un uso intensivo del control de versiones del código fuente. Dado que la lógica de la aplicación dentro de un sistema *RDBMS-only* se almacenará como DBPL, el control de versiones debe abordarse con herramientas de control de versiones disponibles en el RDBMS. Se podría analizar la aplicabilidad de herramientas de control de versiones de esquemas y datos como liquibase [121], DbPatch [122], Flyway [123], Evolve [124] o gitSQL [125]. Estas herramientas podrían requerir adaptaciones para el trabajo real en una arquitectura *RDBMS-only*. Otro problema relevante es la disponibilidad de bibliotecas de uso general. Si bien una tecnología *RDBMS-only* puede producir y consumir servicios web, existen dificultades para operar y realizar tareas específicas debido a la ausencia de bibliotecas que resuelvan problemas cotidianos. Esta dificultad es accidental y puede explicarse por la poca popularidad de la arquitectura.

Se puede argumentar que la dependencia del RDBMS constituye otra desventaja. En teoría, podría realizarse una propuesta similar a partir de un DBPL basado en un lenguaje de propósito general como Java, Perl o Python. Si bien el prototipo de este trabajo utilizó PostgreSQL, y se podrían haber utilizado alguna de esas alternativas, podrían presentar problemas al tratar de trasladar la solución a otros RDBMSs, bien porque no soporten esos lenguajes como DBPL, bien porque tengan particularidades en su implementación, o bien porque en la interacción con el RDBMS se hayan utilizado particularidades de los dialectos del SQL de cada RDBMS. Una propuesta de tecnología *RDBMS-only* que funcione en múltiples RDBMSs probablemente necesitaría una capa específica sobre cada RDBMS soportado que permitiera estandarizar las funciones del *DL Code* a las capas superiores.

Otras dos líneas de investigación interesante quedaron fuera del alcance de este trabajo. La primera es el desarrollo de la capa de *SI Code* para utilizar la arquitectura *RDBMS-only* como proveedor de servicios web. En este caso, sin toda la complejidad de los elementos de la interfaz, podría ser aún más fácil brindar servicios de datos con un enfoque *RDBMS-only*. La segunda es el desarrollo de un IDE que permita el desarrollo de aplicaciones de manera

productiva.

En una interpretación del fragmento de Arquíloco del epígrafe de este capítulo, el erizo y el zorro representan dos formas extremas de personalidades intelectuales. El erizo representa la adherencia a una visión central única, "centrípeta" y finalista; mientras que el zorro representa a quienes, incapaces de percibir un orden único, quedan confinados a percibir los casos particulares, numerosos, diversos e incluso contradictorios en una visión "centrífuga". En las ciencias, los zorros son minoría [126]. Quizá en la tecnología sean mayoría, pero aún podría haber erizos. En algunas posiciones que pretenden defender una arquitectura *database-centric* a toda costa, podría advertirse una personalidad de erizo; pero también podría advertirse una en el sentido opuesto, en otras posiciones que pretenden rechazar *a priori* o ignorar cualquier alternativa que no coincida con su visión central única, su "paradigma" en sentido científico, de orientación a objetos y múltiples niveles. Frente a estos dos erizos opuestos, un zorro podría realizar una petición de tolerancia para la convivencia de "paradigmas tecnológicos".

Si bien creo que este trabajo realiza un pequeño aporte, existen muchas formas de extenderlo, y acaso corregirlo. En primer lugar, los experimentos del Capítulo 6 podrían continuarse tanto para seguir enriqueciendo una aplicación para usuario final, como para incursionar en el problema del desarrollo de un IDE. Los avances que puedan realizarse en estas extensiones, seguramente servirán como retroalimentación tanto para el prototipo tecnológico presentado en el Capítulo 5, como para la propuesta de la arquitectura presentada en el Capítulo 4. En la medida que tanto una propuesta de la arquitectura *RDBMS-only* como la tecnología de los prototipos que puedan desarrollarse siguiéndola evolucionen, tanto más sentido tendrá continuar los análisis comparativos con otras arquitecturas y tecnologías como los presentados en el Capítulo 7. No solo creo posible, sino más bien seguro, que aún deben existir ventajas y posibilidades de una arquitectura *RDBMS-only* no descubiertas en este trabajo.

Bibliografía

[1] Wayne W. Eckerson. "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications." En: *Open Information Systems* 10.1 (1995).

[2] Frederick P. Brooks Jr. "No Silver Bullet Essence and Accidents of Software Engineering". En: *Computer* 20.4 (abr. de 1987), págs. 10-19.

[3] Clay Richardson y John R Rymer. "Vendor landscape: The fractured, fertile terrain of low-code application platforms". En: *Forrester: Cambridge, MA, USA* (2016).

[4] P Vincent y col. "Magic Quadrant for Enterprise Low-Code Application Platforms". En: *Gartner report* 18 (2019).

[5] Ana Nunes Alonso y col. "Towards a polyglot data access layer for a low-code application development platform". En: *arXiv preprint arXiv:2004.13495* (2020).

[6] Edgar Frank Codd. *Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks*. Inf. téc. IBM Research Report RJ 599, ago. de 1969.

[7] Edgar Frank Codd. "A Relational Model of Data for Large Shared Data Banks". En: *Communications of the ACM* 13.6 (jun. de 1970), págs. 377-387.

[8] Edgar Frank Codd. "A Data Base Sublanguage Founded on the Relational Calculus". En: *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) workshop on data description, access and control.* Jul. de 1971, págs. 35-68.

[9] Edgar Frank Codd. "Further Normalization of the Data Base Relational Model". En: *Data Base Systems: Courant Computer Science Symposia Series 6* 6 (1972), págs. 33-64.

[10] Edgar Frank Codd y col. *Relational Completeness of Data Base Sublanguages*. IBM Corporation, mar. de 1972.

[11] Donald D Chamberlin y Raymond F Boyce. “SEQUEL: A structured English query language”. En: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 1974, págs. 249-264.

[12] Theo Haerder y Andreas Reuter. “Principles of Transaction-oriented Database Recovery”. En: *ACM Comput. Surv.* 15.4 (dic. de 1983), págs. 287-317.

[13] Eric C. Cooper. “On the Expressive Power of Query Languages for Relational Databases”. En: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’82. New York, NY, USA: ACM, 1982, págs. 361-365.

[14] Jameela Al-Jaroodi, Nader Mohamed y Junaid Aziz. “Service oriented middleware: trends and challenges”. En: *2010 Seventh International Conference on Information Technology: New Generations*. IEEE. 2010, págs. 974-979.

[15] Joachim W Schmidt y Florian Matthes. “The DBPL project: Advances in modular database programming”. En: *Information Systems* 19.2 (1994), págs. 121-140.

[16] Piero Fraternali, Gustavo Rossi y Fernando Sánchez-Figueroa. “Rich internet applications”. En: *IEEE Internet Computing* 14.3 (2010), págs. 9-12.

[17] Divya Mahajan y col. “In-RDBMS hardware acceleration of advanced analytics”. En: *arXiv preprint arXiv:1801.06027* (2018).

[18] Tim Austwick. “Using Oracle Apex securely”. En: *Network Security* 2013.12 (2013), págs. 19-20.

[19] John Scourias. “Aspects of Client/Server Database Systems”. En: *University of Waterloo* (1995).

[20] G. Alonso y col. *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer Berlin Heidelberg, 2013.

[21] Mario Villamizar y col. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". En: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, págs. 583-590.

[22] Karl Raimund Popper. *Búsqueda sin término*. Tecnos, 2002.

[23] Michael Stonebraker y col. "The End of an Architectural Era: (It's Time for a Complete Rewrite)". En: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, págs. 1150-1160.

[24] Alfonso Vicente, Lorena Etcheverry y Ariel Sabiguero. "An RDBMS-only architecture for web applications". En: *2021 XLVII Latin American Computing Conference (CLEI)*. IEEE. 2021, págs. 1-9.

[25] David Lorge Parnas, Paul C Clements y David M Weiss. "The modular structure of complex systems". En: *IEEE Transactions on software Engineering* 3 (1985), págs. 259-266.

[26] Paul C Clements y Linda M Northrop. *Software Architecture: An Executive Overview*. Inf. téc. Carnegie-Mellon University, Pittsburgh, PA, Software Engineering Institute, 1996.

[27] John A Zachman. "A framework for information systems architecture". En: *IBM systems journal* 26.3 (1987), págs. 276-292.

[28] John F. Sowa y John A. Zachman. "Extending and formalizing the framework for information systems architecture". En: *IBM systems journal* 31.3 (1992), págs. 590-616.

[29] C4ISR Architecture Working Group y col. "C4ISR architecture framework version 2.0". En: *US Department of Defense* (1997).

[30] The Open Group. *TOGAF®, an Open Group standard — The Open Group*. URL: http://www.opengroup.org/subjectareas/enterprise/togaf.

[31] Dewayne E. Perry y Alexander L. Wolf. "Foundations for the Study of Software Architecture". En: *SIGSOFT Softw. Eng. Notes* 17.4 (oct. de 1992), págs. 40-52.

[32] Steven H Spewak y Steven C Hill. *Enterprise architecture planning: developing a blueprint for data, applications and technology*. QED Information Sciences, Inc., 1993.

[33] CIO Council. "Federal enterprise architecture framework version 1.1". En: *Retrieved from* 80 (1999), págs. 3-1.

[34] Philippe Kruchten. "The 4+1 View Model of Architecture". En: *IEEE Softw.* 12.6 (nov. de 1995), págs. 42-50.

[35] Leon Shklar y Richard Rosen. *Web application architecture*. John Wiley & Sons, 2009.

[36] *RFC 3875 - The Common Gateway Interface (CGI) Version 1.1*. URL: https://tools.ietf.org/html/rfc3875.

[37] Jakob Nielsen. "Noncommand user interfaces". En: *Communications of the ACM* 36.4 (1993), págs. 83-99.

[38] Andrew P Black y Jonathan Walpole. "Objects to the rescue! or httpd: the next generation operating system". En: *ACM SIGOPS Operating Systems Review* 29.1 (1995), págs. 91-95.

[39] Won Kim. "Object-Oriented Database Systems: Promises, Reality, and Future." En: *VLDB*. Vol. 19. 1993, págs. 676-692.

[40] Lawrence G Tesler. "Networked Computing in the 1990s". En: *Scientific American* 265.3 (1991), págs. 86-93.

[41] Christopher Ireland y col. "A classification of object-relational impedance mismatch". En: *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*. IEEE. 2009, págs. 36-43.

[42] Bruce J. Neubauer y Dwight D. Strong. "The object-oriented paradigm: more natural or less familiar?" En: *Journal of Computing Sciences in Colleges* 18 (2002), págs. 280-289.

[43] Robert Hirschfeld. "Three-tier distribution architecture". En: *Pattern Languages of Programs (PloP)* (1996), págs. 1-4.

[44] Cory Nance y col. "Nosql vs rdbms-why there is room for both". En: *Proceedings of the Southern Association for Information Systems Conference, Savannah, GA, USA*. 2013.

[45] Craig Larman, Luz María Hernández Rodríguez y Humberto Cárdenas Anaya. *UML y Patrones: Introducción al análisis y diseño orientado a objetos*. Vol. 2. Prentice Hall, 1999.

[46] Philip A Bernstein. "Middleware: a model for distributed system services". En: *Communications of the ACM* 39.2 (1996), págs. 86-98.

[47] Kassem Saleh, Robert Probert y Hassib Khanafer. "The distributed object computing paradigm: concepts and applications". En: *Journal of Systems and Software* 47.2-3 (1999), págs. 125-131.

[48] Richard E Schantz y Douglas C Schmidt. "Middleware for distributed systems: Evolving the common structure for network-centric applications". En: *Encyclopedia of Software Engineering* 1 (2001), págs. 1-9.

[49] Joachim W Schmidt. "Some high level language constructs for data of type relation". En: *ACM Transactions on Database Systems (TODS)* 2.3 (1977), págs. 247-261.

[50] H Eckhardt y col. "Draft report on the database programming language DBPL". En: *Universitat Frankfurt, FR Germany* (1985).

[51] Oracle Corporation. "Oracle Applications with the Network Computing Architecture". En: (1997). URL: http://www.adm.uwaterloo.ca/infofsp/fspupg/techdocs/107nca.pdf.

[52] Oracle Corporation. *An Overview of Oracle Forms Server Architecture*. Inf. téc. Oracle Technical White Paper, abr. de 2000.

[53] Oracle Corporation. *Oracle Forms - Oracle FAQ*. URL: http://www.orafaq.com/wiki/Oracle_Forms.

[54] Piero Fraternali. "Tools and approaches for developing data-intensive Web applications: a survey". En: *ACM Computing Surveys (CSUR)* 31.3 (1999), págs. 227-263.

[55] SV Madhava Krishna y col. "Analysis and Modeling of Evolving Database-centric Web Applications." En: *COMAD*. 2010, pág. 65.

[56] Alastair Monger, Sheila Baron y Jing Lu. "More on Oracle APEX for teaching and learning". En: *the 7th International Workshop on Teaching, Learning and Assessment of Databases, 6 July 2009, University of Birmingham*. 2009, págs. 3-12.

[57] Z Zaharieva y R Billen. *Rapid Development of Database Interfaces with Oracle APEX, used for the Controls Systems at CERN*. Inf. téc. 2009.

[58] Luis Rodríguez Fernández. *Oracle APEX in the CERN Java Cloud*. Abr. de 2015. URL: http://openlab.cern/publications/technical_documents/oracle-apex-cern-java-cloud.

[59] Jiangping Wang y Janet L Kourik. "Delivering database knowledge with web-based labs". En: *ASBBS Proceedings* 19.1 (2012), pág. 923.

[60] Recx Ltd. *Oracle APEX Nessus Plugins*. URL: https://www.recx.co.uk/downloads/RecxApexNessusPlugins.zip.

[61] Philip Greenspun. *Database Backed Web Sites: The Thinking Person's Guide to Web Publishing*. Ziff-Davis Publishing Co., 1997.

[62] Eve Andersson, Philip Greenspun y Andrew Grumet. *Software Engineering for Internet Applications*. The MIT Press, 2006.

[63] John C Shafer y Rakesh Agrawal. "Continuous querying in database-centric web applications". En: *Computer networks* 33.1 (2000), págs. 519-531.

[64] K. Selçuk Candan y col. "Enabling Dynamic Content Caching for Database-driven Web Sites". En: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. New York, NY, USA: ACM, 2001, págs. 532-543.

[65] Jan Ploski y col. "Introducing version control to database-centric applications in a small enterprise". En: *IEEE software* 24.1 (2007).

[66] Torsten Grust, Nils Schweinsberg y Alexander Ulrich. "Functions are data too: defunctionalization for PL/SQL". En: *Proceedings of the VLDB Endowment* 6.12 (2013), págs. 1214-1217.

[67] Torsten Grust y Alexander Ulrich. "First-class functions for first-order database engines". En: *arXiv preprint arXiv:1308.0158* (2013).

[68] Alvin Cheung, Samuel Madden y Armando Solar-Lezama. "Sloth: Being lazy is a virtue (when issuing database queries)". En: *ACM Transactions on Database Systems (ToDS)* 41.2 (2016), págs. 1-42.

[69] Toon Koppelaars. "A Database-Centric Approach to J2EE Application Development". En: *Technical Report* (2014).

[70] Toon Koppelaars. *The Helsinki Declaration (IT-version): @Hotsos 2009: Starting this blog*. Mar. de 2009. URL: http://thehelsinkideclaration.blogspot.com.uy/2009/03/start-of-this-blog.html.

[71] Julie Smith David, David Schuff y Robert St. Louis. "Managing your total IT cost of ownership". En: *Communications of the ACM* 45.1 (2002), págs. 101-106.

[72] Bryn Llewellyn. *NoPlsql versus ThickDB*. URL: https://blogs.oracle.com/plsql-and-ebr/noplsql-versus-thickdb.

[73] Bryn Llewellyn. *Why use PL/SQL?* URL: https://www.oracle.com/a/tech/docs/why-use-plsql-whitepaper-10.pdf.

[74] *Kscope17*. URL: http://kscope17.com/.

[75] Marius Muji. "Application Development in Database-Driven Information Systems". En: *Acta Universitatis Sapientiae-Electrical & Mechanical Engineering* 2 (2010).

[76] *MDA — Object Management Group*. URL: http://www.omg.org/mda/.

[77] Octavian Andrei Dragoi. "The conceptual architecture of the Apache web server". En: *Dept. of Computer Science, University of Waterloo* (1999), págs. 1-10.

[78] Oracle Corporation. *Database PL/SQL Language Reference*. URL: https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/wrap.htm#LNPLS016.

[79] Pete Finnigan. *How to unwrap PL/SQL*. 2009. URL: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Finnigan.pdf.

[80] Oracle Corporation. *It is illegal to unwrap the APEX code for educational purposes? — Oracle Community*. 2017. URL: https://community.oracle.com/thread/4027528.

[81] Oracle Corporation. *Oracle Application Express Documentation*. URL: https://docs.oracle.com/database/apex-5.1/.

[82] Oracle Corporation. *Oracle Application Express Architecture*. URL: http://www.oracle.com/technetwork/developer-tools/apex/apex-arch-086399.html.

[83] Oracle Corporation. *Application Express App Builder User's Guide.* URL: https://docs.oracle.com/database/apex-5.1/HTMDB/toc.htm.

[84] *Oracle Application Express - Collateral.* URL: http://www.oracle.com/technetwork/developer-tools/apex/learnmore/apex-presentations-1866619.html.

[85] Oracle Corporation. *Espacio: Oracle Application Express (APEX) — Oracle Community.* URL: https://community.oracle.com/community/database/developer-tools/application_express.

[86] Oracle Corporation. *Application Express App Builder User's Guide.* URL: https://docs.oracle.com/database/apex-5.1/HTMDB/utilizing-debug-mode.htm#HTMDB10003.

[87] Oracle Corporation. *Oracle APEX Platform.* URL: https://apex.oracle.com/en/platform/.

[88] *Understanding mod_plsql.* URL: https://docs.oracle.com/middleware/11119/classic/use-modplsql/concept.htm.

[89] Dimitri Gielis. *Moving to the APEX Listener.* https://www.slideshare.net/DimitriGielis/moving-to-the-apex-listener. 2015.

[90] Oracle Corporation. *Apache PL/SQL Gateway Module.* URL: https://oss.oracle.com/projects/mod_owa/dist/documentation/modowa.htm.

[91] Torsten Zimmermann y col. "How HTTP/2 pushes the web: An empirical study of HTTP/2 server push". En: *2017 IFIP Networking Conference (IFIP Networking) and Workshops.* IEEE. 2017, págs. 1-9.

[92] Roberto Gonzalez y col. "The cookie recipe: Untangling the use of cookies in the wild". En: *2017 Network Traffic Measurement and Analysis Conference (TMA).* IEEE. 2017, págs. 1-9.

[93] Oracle Corporation. *Understanding Session State Management.* https://docs.oracle.com/database/121/HTMDB/concept_ses.htm.

[94] Oracle Corporation. *Creating a Form.* https://docs.oracle.com/database/apex-18.1/HTMDB/using-a-wizard-to-create-a-form.htm.

[95] Oracle Corporation. *Creating a Report Using the Create Page Wizard*. https://docs.oracle.com/database/apex-18.1/HTMDB/creating-report-create-page-wizard.htm.

[96] *PostgREST*. URL: https://postgrest.org/.

[97] L. Masinter. *RFC2397 - The "data"URL scheme*. 1998. URL: https://tools.ietf.org/html/rfc2397.

[98] PgBouncer Authors. *PgBouncer*. URL: https://www.pgbouncer.org.

[99] Tatsuo Ishii. *pgpool Wiki*. https://www.pgpool.net.

[100] Junwen Yang y col. "How not to structure your database-backed web applications: a study of performance bugs in the wild". En: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, págs. 800-810.

[101] Transaction Processing Performance Council (TPC). *TPC BENCHMARK W (Web Commerce) Specification*. Dic. de 2003. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpcw_v2.0.0.pdf.

[102] Daniel A Menascé. "TPC-W: A benchmark for e-commerce". En: *IEEE Internet Computing* 6.3 (2002), págs. 83-87.

[103] Harold W. Cain y Ravi Rajwar. "An architectural evaluation of Java TPC-W". En: *In Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*. 2001, págs. 229-240.

[104] Todd Bezenek y col. *Java TPC-W Implementation Distribution*. 2011. URL: https://pharm.ece.wisc.edu/tpcw.shtml.

[105] Daniel A Menascé y col. "A methodology for workload characterization of e-commerce sites". En: *Proceedings of the 1st ACM conference on Electronic commerce*. 1999, págs. 119-128.

[106] Zhen Ming Jiang, Ahmed E Hassan y Richard C Holt. "Visualizing clone cohesion and coupling". En: *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*. IEEE, 2006, págs. 467-476.

[107] Tae Kyung Sung. "E-commerce critical success factors: East vs. West". En: *Technological forecasting and social change* 73.9 (2006), págs. 1161-1177.

[108] ISO/IEC. *ISO/IEC 25010 System and software quality models*. Inf. téc. ISO/IEC, 2010.

[109] The Kubernetes Authors. *Pods — Kubernetes*. https://kubernetes.io/es/docs/concepts/workloads/pods/.

[110] 2ndQuadrant Ltd. *Replication & Failover Manager for Postgres*. https://www.2ndquadrant.com/en/resources/repmgr/.

[111] EBD. *EDB. Power to Postgres*. https://www.enterprisedb.com/.

[112] Edward A Luke. "Defining and measuring scalability". En: *Proceedings of Scalable Parallel Libraries Conference*. IEEE, 1993, págs. 183-186.

[113] André B. Bondi. "Characteristics of Scalability and Their Impact on Performance". En: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. event-place: Ottawa, Ontario, Canada. New York, NY, USA: ACM, 2000, págs. 195-203. ISBN: 1-58113-195-X.

[114] Raúl Pena-Ortiz y col. "Providing TPC-W with web user dynamic behavior". En: *CLEI electronic journal* 15.2 (2012), págs. 2-2.

[115] Ronald C. Dodge JR, Daniel A. Menascé y Daniel Barbara. "Testing E-Commerce Site Scalability With Tpc-W". En: *In Proc. of 2001 Computer Measurement Group Conference*. 2001, págs. 457-466.

[116] *Network Monitor, Firefox Developer Tools*.

[117] Oracle Corporation. *Oracle Store*. URL: https://shop.oracle.com/apex/f?p=dstore:home:0.

[118] Karl R. Popper. *La Miseria del historicismo*. 1a ed. (Area del conocimiento. Humanidades. Madrid: Taurus, 2010.

[119] Tim Berners-Lee. *Information Management: A Proposal*. Inf. téc. CERN, 1989. URL: http://www.w3.org/History/1989/proposal.html.

[120] Thomas S Kuhn. *La estructura de las revoluciones científicas*. Fondo de cultura económica, 2019.

[121] *Liquibase*. URL: https://www.liquibase.org/.

[122] *DbPatch*. URL: https://github.com/dbpatch/DbPatch.

[123] *Flyway*. URL: https://flywaydb.org/.

[124] *Evolve*. URL: https://evolve-db.netlify.app/.

[125] *gitSQL*. URL: https://gitsql.net/.

[126] Isaiah Berlin. *El erizo y el zorro*. Península, 2016.